ROLE-BASED AND AGENT-ORIENTED

TEAMWORK MODELING

A Dissertation

by

SEN CAO

Submitted to the Office of Graduate Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

August 2005

Major Subject: Computer Science

ROLE-BASED AND AGENT-ORIENTED

TEAMWORK MODELING

A Dissertation

by

SEN CAO

Submitted to the Office of Graduate Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

Approved by:

Chair of Committee,	Richard A. Volz
Committee Members,	Thomas R. Ioerger
	Jennifer L. Welch
	Amarnath Banerjee
Head of Department,	Valerie E. Taylor

August 2005

Major Subject: Computer Science

ABSTRACT

Role-Based and Agent-Oriented
Teamwork Modeling. (August 2005)
Sen Cao, B.S., Shanghai Maritime University, China;
M.S., Institute of Software, Chinese Academy of Sciences, China
Chair of Advisory Committee: Dr. Richard Volz

Teamwork has become increasingly important in many disciplines. To support teamwork in dynamic and complex domains, a teamwork programming language and a teamwork architecture are important for specifying the knowledge of teamwork and for interpreting the knowledge of teamwork and then driving agents to interact with the domains. Psychological studies on teamwork have also shown that team members in an effective team often maintain shared mental models so that they can have mutual expectation on each other. However, existing agent/teamwork programming languages cannot explicitly express the mental states underlying teamwork, and existing representation of the shared mental models are inefficient and further become an obstacle to support effective teamwork. To address these issues, we have developed a teamwork programming language called Role-Based MALLET (RoB-MALLET) which has rich expressivity to explicitly specify the mental states underlying teamwork. By using roles and role variables, the knowledge of team processes is specified in terms of conceptual notions, instead of specific agents and agent variables, allowing joint intentions to be formed and this knowledge to be reused by different teams of agents. Further, based on roles and role variables, we have developed mechanisms of task decomposition and task delegation, by which the knowledge of a team process is decomposed into the knowledge of a team process for individuals and then delegate it to agents. We have also developed an efficient representation of shared mental models called Role-Based Shared Mental Model (RoB-SMM) by which agents only maintain individual processes complementary with others' individual process and a low level of overlapping called team organizations. Based on RoB-SMMs, we have developed two

reasoning mechanisms to improve team performance, including Role-Based Proactive Information Exchange (RoB-PIE) and Role-Based Proactive Helping Behaivors (RoB-PHB). Through RoB-PIE, agents can anticipate other agents' information needs and proactively exchange information with them. Through RoB-PHB, agents can identify other agents' help needs and proactively initialize actions to help them. Our experiments have shown that RoB-MALLET is flexible in specifying reusable plans, RoB-SMMs is efficient in supporting effective teamwork, and RoB-PHB improves team performance.

To my parents and to Ping

ACKNOWLEDGMENTS

I would like to take this opportunity to thank my advisor, Richard Volz. His unique combination of sharp insight, quick wit and patient guidance made this research endeavor full of challenges and fun. Dr. Volz has taught me well; especially that teamwork modeling may not be just a mechanism for agents, but a way of human lives. I would like to thank Dr. Thomas Ioerger, who inspired so many exciting research discussions and gave me many useful suggestions. I would like to thank the rest of my committee members, Dr. Jennifer Welch, and Dr. Amarnath Banerjee for their taking time in reviewing this dissertation and giving valuable comments.

I have grown academically and personally from interactions with many other people from the group of MURI project across Texas A&M University, Wright State University and Pennsylvania State University. I would like to thank Dr. John Yen and Dr. Wayne L. Shebilske for sharing their ideas in the group discussions of MURI project. I would like to thank Michael Miller for his exciting discussions with me and especially his great help for my experiments in this research. I would like to thank Jamison Johnson, Maitreyi Nanjanath, Jonathan Whetzel, Dianxiang Xu, Yu Zhang, Linli He, Norman Ma, Jesse Plymale for cooperative works and fun moments. My thanks also go to Barbara Barby, Stacy Stokes and Ann Hart for their help.

Last, but never least, I would like to thank my wife, Ping Ding, and our parents for their support, their encouragement and their love throughout all these years.

This research was funded by DoD MURI grant F49620-00-1-0326 administered through AFOSR.

TABLE OF CONTENTS

CHAPTER Page

CHAPTER Page

Page

LIST OF FIGURES

Page

Page

Page

Page

Page

LIST OF TABLES

CHAPTER I
INTRODUCTION

I.1 Motivation

In recent years, teamwork has become increasingly important in many disciplines, such as business management, sports and entertainments, defense simulations, and virtual training. A great amount of research has been done to foster effective teamwork. Many researchers have been striving to develop teamwork theories and architectures to facilitate multiple agents to work closely as a team in complex and dynamic domains, in which agents are autonomous, heterogeneous distributed over various platforms. However, ensuring that agents work as an effective team remains a very difficult challenge. In particular, recent research in this area is directed toward facilitating a mixture team of humans and agents [69, 70, 127].

Teamwork theories, such as Cohen and Levesque's joint intention theory [22, 59], Grosz and Kraus' shared plan theory [36, 37] and Jennings' joint responsibility [47, 50], have explored the critical mental states underlying teamwork driving agents to act together as a team and the interactions leading their individual actions to team efforts. The critical mental states include operators, mutual beliefs, shared goals, team plans, and joint intentions. Through interactions, agents transform their team mental states (such as joint goals, joint intentions and team plans) to individual mental states (such as individual goals, individual intentions and individual operator/plan); and the evolution of mental states eventually leads to the individual actions. For example, joint intention theory [22, 59] formalized the transformation from joint intentions and joint commitments to individual intentions; shared plan theory [36, 37] elaborated the evolution from shared plans to individual actions; the cooperative problem solving related to joint responsibility in [120] suggested a process from problem recognition, to planning

---

This thesis follows the style of Artificial Intelligence.

a course of actions, to forming a team to execute the actions, and to eventually having agents complete the actions.

To apply these teamwork theories to simulate teamwork, a teamwork language is demanded to explicitly express the mental states underlying teamwork. Particularly in complex and dynamic domains, many unexpected uncertainties could occur, such as dynamic changes in team's goals, team members' unexpected failures to fulfill their responsibilities, dynamic decision-making in dynamic environment, and dynamically backing up other team members. The decisions on handling these uncertainties are tightly related to the mental states underlying teamwork. To specify the knowledge about handling these uncertainties, it is more important for the teamwork language to explicitly express the mental states underlying teamwork.

Current teamwork languages (or scripts in teamwork architectures), such as STEAM [102, 103], GRATE [49, 53], dMARS [25] and JACK [10] languages are designed to express some of those mental states so as to enable certain features of their teamwork architectures; however, they cannot explicitly express all of them. STEAM is based on SOAR, which basically lacks the explicit expression of team mental states, such as shared goals and joint intentions. STEAM stimulates teamwork by creating a basic building block of joint intention. GRATE [49, 53] indirectly expresses joint intentions through joint commitments and conventions, which are represented as generic rules in agent models. Although dMARS [25], a more recent version of PRS, can explicitly express individual mental states such as (beliefs, goals and intentions), it still lacks the explicit constructs for team mental states (such as shared goal and joint intentions). JACK can express the mental states underlying teamwork, but they are indirectly specified in Java by a number of additional constructs, such as *agent*, *plan*, and *team*. As later discussed in Chapter II, teamwork theories, architectures, and systems highly rely on the expression of the mental states underlying teamwork. A general teamwork language with explicit expressivity of the mental states underlying teamwork will give

its users great freedom to utilize existing teamwork mechanisms/methods and explore new teamwork mechanisms/methods.

Considering the perspective of software engineering, the teamwork language would better allow to specifying teamwork knowledge conceptually for being reused, particularly, team plans are better specified in terms of abstract entities, instead of specific agents, so as to be reused by different teams of agents.

Therefore, the teamwork research community still needs a teamwork language which has rich expressivity to specify the critical mental states underlying teamwork explicitly and allows dynamic team structures.

Corresponding to the needed teamwork language, a teamwork architecture is also needed which can interpret the knowledge (including the mental states) in the teamwork language, transform team mental states into individual mental states and eventually into individual actions, and allow incorporating various reasoning mechanisms, such as information exchange and helping behaviors. Considering that team plans are specified in terms of some abstract entities, the teamwork architecture should contain a certain mechanism that builds the connections between the abstract entities used in a team plan and the agents invoking the team plan. Moreover, this mechanism should be compatible with the mental states and interactions underlying teamwork, particularly the evolution from team mental states to individual mental states and eventually to individual actions.

In recent years, shared mental models have been used in a teamwork architecture, CAST [127, 128, 129], to maintain team mental states and to coordinate individual actions. However, maintaining completely shared mental models among agents caused a great amount of unnecessary communication and this can be an obstacle to teamwork effectiveness. Developing efficient shared mental models is still a difficult challenge. Moreover, using shared mental models in the teamwork architecture allows incorporating various reasoning mechanisms to improve team performance, such as information exchange and helping behaviors. To utilize shared mental models in the teamwork architecture, shared mental models must be compatible with the evolution of mental states.

I.2 Research Approach and Contributions

The goal of this research is to develop methods for using the concept of complementary and minimally overlapping shared mental models as a basis for agent teamwork, enabling task decomposition, task delegation, dynamic planning and helping behaviors. Moreover, we design a flexible teamwork architecture, RoB-CAST (Role-Based Collaborative Agents for Simulating Teamwork), for supporting a team of agents in complex and dynamic domains. The central thesis of this research is that

> *Our approach can represent and simulate flexible and effective teamwork in which team plans are specified in terms of roles and role variables, the critical mental states underlying teamwork can be explicitly specified, agents only maintain distributed mental models that are largely complementary with only a low level of overlapping (RoB-SMMs - Role-Based Shared Mental Models), and proactive role-based information exchange and proactive helping behaviors are fostered.*

Our research approach is to first give the basic syntax of a teamwork language that allows explicitly specifying the mental models underlying teamwork, and the semantics of the constructs in terms of the mental states. Both individual knowledge (including agents, agents' capabilities, individual beliefs, individual goals, individual intentions and individual operators) and team knowledge (including team, mutual beliefs, shared goals, joint intentions, and team operators/plans) can be specified. The semantics of the constructs delineates the relationships among knowledge. In particular, individual/joint intentions are specified by individual/shared goals or by individual/team operator/plans; the preconditions of an operator or plan are evaluated based on individual/mutual beliefs depending on the operator/plan is individual or team operator/plan; the effects of an operator/plan reflect the change of individual/mutual beliefs after its performance; the conditions in flow constructs (e.g., IF and WHILE constructs) are evaluated based the individual/mutual beliefs of the agents involved in the actions in the flow constructs.

Second, we introduce three formal concepts, position, role and role variable, so that team plans, called role-based plans, are specified in terms of these concepts instead of

specific agents. Unlike the informal concept of role used in other teamwork architectures, such as the roles used in STEAM [103, 104] and TOP [94, 106] and the variant of role (agent variable) used in MALLET [28], the concepts of position, role and role variable are formalized based on the classificatory concepts in role theory literature that capture the behavioral aspects of role [4]. According to their formalizations, we include the syntax to declare positions, roles and role variables in the teamwork language, and clarify the semantics of the syntax.

A particularly important construct in the language is the role-based plan. This is specified in terms of a virtual team of roles and a set of role variables. The actions associated with roles and role variables are specified in the process of the role-based plan. The temporal ordering on the actions is specified by the constructs of flow control. Moreover, a role-based plan contains a set of constraints on delegating the roles in the plans to agents. Before a team of agents actually executes a role-based plan, each role in the plan needs to be delegated to an agent; and the delegations of all roles to the agents must satisfy the constraints on delegating the roles in the plans to agents. During the plan execution, a role is dynamically selected to fill a role variable; and the selected role must satisfy the constraints on the selection. One of the important differences between RoB-MALLET and MALLET is that RoB-MALLET can distinguish static and dynamic constraints through roles and role variables respectively, while MALLET only uses Agent-bind constructs through agent variables. We use a CSP (Constraint Satisfaction Problem) algorithm to search an admissible team assignment for a team of agents to execute a role based plan and to select a role for a role variable selection. In addition, we show how role-based plans could be used as actions together with individual operators in planning algorithms (either forward or backward state-space search), and give a simple version of planning algorithm that just searches a role-based plan for a team of agents, without decomposing a goal to a sequence of states and using operators.

Third, we develop the mechanisms for task decomposition and task delegation, which are compatible with the evolution of mental states and practically lead to individual actions, based on role-based plans. Without loss of generality, a task of a team

of agents is that the agents invoke a role-based plan. To enable our mechanisms of task decomposition and task delegation, we define a notion of responsibility in terms of what a responsibility contains and how a responsibility impacts the mental states of the agent(s) taking the responsibility. The team responsibility of all the roles in a role-based plan contains all temporally ordered actions associated with the roles and their impact on the mental states of the agents (*T*) taking the team responsibility. The individual responsibility of a role (or role variable) contains the temporally ordered actions associated with the role (or role variable) and their impact on the mental states of the agent taking the responsibility.

We introduce a graphic language to represent what a responsibility contains. To decompose a task (i.e., an invocation of a role-based plan) into individual tasks, we translate a role-based plan into a team responsibility graph and then decompose the team responsibility into individual responsibility graphs. As long as the role-based plan is hierarchical (i.e., containing sub-plan invocations), the individual responsibilities of roles at the high level dynamically expand the sub-plan invocations by the responsibilities of roles at the low level delegated to the agents (to which the roles at the high level are delegated). To delegate tasks to agents, we use the CSP algorithm to search for an admissible team assignment for a team of agents to execute a role-based plan and to select a role for a role variable selection. Consequently, the individual responsibilities of the roles and role variables in the role-based plan are delegated to agents invoking the role-based plan.

Fourth, we develop an efficient representation of role-based shared mental models (*RoB-SMMs*) that are complementary, distributed, and have only a low level of overlapping by taking the advantage of the mechanisms of task decomposition and task delegation. In the *RoB-SMM* of an agent (*ag*), a team process is maintained by a team organization and an individual process. The team organization is a hierarchy of shared goals, the role-based plans to achieve the goals, and the team structures for agents to execute the plans (i.e., the team assignments and role variable selections for executing the plans). An individual process only contains the portion of team process related to the

agent. It is represented by an extension of Petri Net, called *RoB-CAST-PN*. We design an algorithm to translate an individual responsibility into a *RoB-CAST-PN* representation and an algorithm to dynamically compose a *RoB-CAST-PN* so as to dynamically load/unload the *RoB-CAST-PN* representations corresponding to the responsibilities of the roles delegated to an agent into/from the agent's individual process. Each agent executes its individual process. We show how the executions of individual processes are coordinated and how team organizations interact with individual processes. In *RoB-SMMs*, a team process is distributed among the team of agents by each agent maintaining a team organization and an individual process; the individual processes are complementary to each other; and the overlapping of *RoB-SMMs* is only in the agents' team organization. Using *RoB-SMMs* can dramatically reduce the communication for maintaining shared mental models, increase the concurrency of executing team processes, and further improve team performance.

Last, based on *RoB-SMMs*, we develop two reasoning mechanisms, including RoB-PIE (Role-Based Proactive Information Exchange) and RoB-PHB (Role-Based Proactive Helping Behaviors), to improve team performance. The mechanism of PIE (Proactive Information Exchange) was originally suggested based on shared mental models in CAST [127], which requires a great amount of communication to maintain shared mental models. We re-formalize PIE to RoB-PIE based on *RoB-SMMs*, which requires a much smaller amount communication to maintain shared mental models. Moreover, our RoB-PIE can facilitate proactive information exchange across the boundaries of simultaneous tasks (i.e., plans at the top level). A set of algorithms is designed to implement RoB-PIE. Based on *RoB-SMMs*, we also formalize RoB-PHB to facilitate proactive helping behaviors among agents, including identifying help needs, and determining what actions and which teams could perform the actions so as to cover the help needs. A set of algorithms is designed to implement RoB-PHB. Among the algorithms of RoB-PHB, an offline algorithm extracts potential help needs from role-based plans; an online algorithm identifies actual help needs based on *RoB-SMMs*, and

searches role-based plans to cover the help needs, and forms teams to actually execute the role-based plans.

The novel contributions of this research are: 1) We give a teamwork language, RoB-MALLET, to allow explicit specification of the mental states underlying teamwork, and introduce three formalized concepts of position, role and role variable to define role-based plans so that role-based plans can be reused by different teams of agents and be used in planning algorithms. 2) We introduce mechanisms of task decomposition and task delegation based on role-based plans, and define a notion of responsibility so that the evolution of mental states can accommodate the mechanisms of task decomposition and task delegation and eventually lead to individual actions. 3) We introduce an efficient representation of *RoB-SMMs*, which is complementary, distributed, and with a low level of overlapping. Using *RoB-SMMs* can dramatically reduce the communication for maintaining shared mental models, increase the concurrency of executing team processes, and further improve team performance. 4) We re-formalize a mechanism of RoB-PIE. RoB-PIE is based on *RoB-SMMs*, which only require a small amount of communication to maintain shared mental models, and RoB-PIE can facilitate proactive information exchange across the boundaries of simultaneous plan invocations (at the top level). 5) We formalize a mechanism of RoB-PHB. Based on *RoB-SMMs*, agents can identify other agents' help needs and proactively provide helping behaviors to the agents.

I.3 Structure of the Dissertation

This chapter provides an overview of the dissertation, including the motivation of this research and the overall idea of our approach and contributions. The rest of this dissertation is organized as the following:

- In Chapter II, we briefly review previous work related to this research.
- In Chapter III, we give the syntax of RoB-MALLET, explain how the mental states underlying teamwork can be explicitly expressed in RoB-MALLET, and

elaborate the semantics of RoB-MALLET in terms of the mental states underlying teamwork.

- In Chapter IV, 1) we formalize the concepts of position, role, and role variable; 2) we define a notion of role-based plans in terms of positions, roles and role variables, give the syntax and semantics related to position, role, role variable and role-based plan, and explain how actions in role-based plans are specified; 3) we define a notion of responsibility, including individual responsibility of a role, individual responsibility of a role variable and team responsibility, introduce a graphic language of responsibility, and present algorithms of translating a role-based plan to a team responsibility graph and algorithms of decomposing a team responsibility graph into individual responsibility graphs; 4) we introduce the delegation of roles to agents, formalize a notion of admissible assignment, and present a CSP algorithm to search for admissible assignments; and 5) we explain how role-based plans could be used in planning algorithms and present an algorithm of searching for a role-based plan for a team of agents to achieve a goal.
- In Chapter V, 1) we introduce *RoB-SMMs* and the knowledge in *RoB-SMMs*; 2) we describe how team processes in *RoB-SMMs* are maintained by team organizations and individual processes; 3) we formalize an extension of Petri Nets, RoB-CAST-PN, to represent individual processes, and present algorithms of translating an individual responsibility to a RoB-CAST-PN and algorithms of dynamically composing individual processes; 4) we describe the interaction between team organization and present an algorithm of executing individual process; 5) we re-formalize RoB-PIE based on RoB-SMMs and present the algorithms to implement RoB-PIE; we formalize RoB-PHB based on RoB-SMMs and present the algorithms to implement RoB-PHB.
- In Chapter VI, we describe the RoB-CAST architecture and its components, introduce the domain and measures for evaluating RoB-CAST, and design and analyze three sets of experiments we have run in the domain.

- Finally, in Chapter VII, we make our conclusions and discuss some future directions.

CHAPTER II
RELATED WORK

The methods presented in this dissertation are built on several areas of previous research, including teamwork theories, teamwork architectures (including the agent/team programming languages used if there are), role, and shared mental models. In this chapter, we will outline related work in these categories.

II.1 Teamwork Theories

An agent is autonomous on controlling its internal states and actions on its environment [121]. While an individual action is performed by an agent to transit one state to another [31, 32], an action performed by a team of agents, called joint, collective, group, or social action [36, 109, 114, 119], is more than the union of simultaneous and coordinated individual actions. Teamwork theories mainly concern the properties of joint actions and mathematically formalize the representation, evolution and reasoning of joint actions, particularly the shared mental states driving team members to act together and the interaction leading their individual actions to team efforts. The most representative teamwork theories include Cohen and Levesque's joint intention theory [20, 22, 23, 59], Grosz and Kraus' shared plan theory [35, 36, 37] and Jennings' joint responsibility [47, 50, 52, 120].

II.1.1 Joint Intention Theory

Joint intention theory was developed based on Bratman's original hypotheses on intentions, that intention is a critical functional mental state of determining individuals' actions and persisting through their attempt to accomplish the actions [5]. In order for agents to perform joint actions, certain shared knowledge (i.e., intentions) should be maintained by the agents. The relationship of agents' intentions is discussed in [5]. The consistency among agents' intentions is discussed in [6, 7, 8]. Searle [88] and Tuomela [113, 110, 111, 112] suggested that joint intention, which extends intention to the group context, is crucial for a group to accomplish joint actions. Searle [88] explained that

certain shared knowledge of driving agents to perform group actions should be maintained by the individuals in the group. Tuomela [113, 110, 111, 112] distinguished joint intentions from individual intentions in two main aspects: 1) joint intentions require agents to make agreement before satisfying the joint intentions; 2) agents satisfy the joint intentions by cooperative achievement process.

Cohen and Levesque [20] formalized the relationships among an agent's beliefs, goals, actions and intentions. The essential property of intention is captured by specifying how an agent is committed to its goal and the explicit condition under which it can drop the goal. Their joint intention theory [22, 59], which extends the formalism of individual intention to the team context, is based on logic formalism, called joint persistent goal. An team of agents has a joint persistent goal relative *q* to achieve *p* iff: 1) all agents mutually believe that *p* is currently false; 2) all agents know that they all want *p* to be true eventually; and 3) (2) until all agents mutually believe that *p* is true, that *p* will never be true, or that *q* is false, they mutually believe that they each have *p* as a weak achievement goal relative q^1. To coordinate agents' teamwork, joint commitment and joint intention are maintained by all the agents. A team of agents has a joint commitment to a goal *g* iff the agents have a joint persistent goal to achieve *g*. A team of agents has a joint intention to an action *a* iff the agents have a joint persistent goal of having action *a* done. Moreover, the evolution from joint commitments and joint intentions to individual commitments and intentions is formalized. By deriving individual intention from joint intention, each agent can refer how it can perform teamwork by performing individual actions. A set of communicative primitives is designed to establish and discharge joint commitment and joint intention and to maintain mutual beliefs among all team members [21, 92]. Mutual beliefs play critical roles in their joint intention theory. The formalisms of joint persistent goal, joint commitments, and joint intentions are derived based on mutual beliefs. Agents need to maintain mutual

[1] Agent *ag* has a weak achievement goal relative to *q* and with respect to a team to achieve *p* if one of the following conditions holds:

1. Agent *ag* does not believe that p is true, and has a goal to eventually make p true;
2. Agent *ag* believes that *p* is true, that *p* will never be true, or that *q* is false, but has a goal that the status of *p* be mutually believed by all the team members.

beliefs so as to maintain these critical mental states. For example, suppose a team of agents has a joint intention to do an action, one of the agents believes that the action cannot be accomplished. However, the agent cannot drop the intention by itself. The agent needs to make all agents mutually believe that the action cannot be accomplished, and then all drop the intention together.

II.1.2 Shared Plan Theory

Grosz and Kraus [35, 36, 37] suggested that collaborative activity must be rested eventually in the actions of individual agents but is more complex than the aggregate of individual actions. They urged that joint actions differ from individual actions by each agent having partial knowledge, the need to form mutual commitment, the assessment of the capabilities of all involved agents, and each agent having partial intention. Thus, their shared plan theory, as a formalism of joint actions, treats not only the intentions, ability, and knowledge about actions of individual agents, but also their coordination in group planning and acting; also, shared plans may be incrementally formed and executed by the agents [36, 37]. Agents may dynamically reconcile their intentions with incremental formation and dynamic execution of shared plans [39, 98, 99].

Pollack [77, 78] viewed plans as complex mental attitudes, by which plans entail beliefs and intentions of agents. She suggested that plan reference of analyzing the mental state of the plan itself betrays cooperative communication among agents. She also suggested that plans are used not only to guide actions of agents, but also to control reasoning of agents and to facilitate cooperation among agents [76]. The formalization of shared plans [36, 37] extended Pollack's mental state model of plans [77, 78] to the situation in which multiple agents together form a plan to perform a collaborative action of the agents. The formulation of shared plans adjusts mental states for agents to coordinate their activities.

The formalization of shared plans [77, 78] provides mental-state specifications of both shared plans and individual plans. Unlike Cohen and Levesque's joint intention that requires specific actions for achieving goals, the formalization of shared plans introduce "*intend-to*" to represent the propositional attitude of *intending to* do an action and

"*intend-that*" to represent the attitude of *intending that* a proposition hold. While actions and plans in Cohen and Levesque's joint intention are complete (i.e., all recipes are pre-defined) and only at high-level of abstraction, a shared plan could be either a full shared plan (FSP) or a partial shared plan (PSP) because agents may have either partial or complete beliefs and intentions. In a partial shared plan, 1) some sub-actions may have not deployed to any agent; 2) the agents may have only partial recipe for doing an action; 3) the agents may have only partial shared plan for doing some of subsidiary actions in the recipe; or 4) the agents may have only partial shared plan for some of contracting actions. PSP does not require that a plan is complete before agents assuming the plan. Agents can evolve PSP to FSP by planning with the execution. Therefore, the formalization of shared plans is more suitable for the agents in the complex and dynamic environments.

II.1.3 Joint Responsibility

By taking a philosophical hypothesis that all coordination mechanisms can ultimately be reduced to (joint) commitments and their associated conventions, Jennings formalized joint responsibility based on commitment and convention [47, 48, 50, 51, 52]. According to [50], commitments are viewed as pledges to undertake a specified course of actions while conventions provide a means of monitoring commitments in changing circumstances. Joint persistent goal, which is the underlying concept of joint intention, can be derived from commitment and convention and further joint intention [22, 59]. Moreover, joint commitments and conventions are used in a procedure of cooperative problem solving [48, 120] to find solutions (i.e., courses of actions to achieve a goal) and team formation (i.e., who can contribute to the performance of the courses of actions). In [47], a joint responsibility of a group of agent for achieving a goal *g* is defined as that the agents mutually believe that, 1) the agents have a joint persistent goals to achieve the goal *g*, 2) the agents have a solution, and 3) the agents have a team formation. Moreover, by having a joint responsibility, agents are mutually aware of the conditions under which cooperative problem solving can commence, how

individual agents should behave once cooperative problem solving has begun and minimum conditions that agents must satisfy.

Rather than focus on breaking team mental states (joint responsibility) into individual mental states (individual actions), Ioerger's and Johnson's formal responsibility [43], based on mutual beliefs and goals, captures collaborative behaviors in a teams, particularly delegation relations between agents. Their formalism concentrates on two central features of responsibility: persistence and dependency. Persistence means that, if an agent has a responsibility of Θ (either a goal or an action), the agent has a persistent goal to have done, or delegate Θ to another agent, or Θ becomes unachievable. Dependency means that the responsibility is assigned by a delegating agent who wants it done. To maintain the dependency of a responsibility, agents are required to maintain a mutual belief on the status of the goal of the responsibility, as well as the need by the superior (the agent delegating the responsibility), and the achievability by the subordinate (the agent to which the responsibility is delegate). This formalism provides a mechanism to achieve robust teamwork. For example, if an agent fails, the agent can delegate its responsibility to another agent.

II.2 Teamwork Architectures

Many teamwork architectures have been developed to support teamwork. Different teamwork architectures may use different agent languages to specify agents' knowledge and focus on supporting different properties of teamwork. In this section, we briefly introduce several representative teamwork architectures, including BDI [82, 83], PRS and dMARS [25, 26, 73], TRL [44], TOP [64, 94, 106, 107, 108], JACK [45, 12], RETSINA [34, 90, 100], STEAM [80, 101, 102, 103, 105], GRATE [47, 49, 52, 53], and CAST [127, 128, 129].

Rao and Beorgeff's [82, 83] Belief-Desire-Intention architecture (BDI), which is the most well-known multi-agent architecture, emphasizes practical reasoning about the current beliefs and goals, and then determining the best possible actions. Unlike joint intention theory that forms intentions based on beliefs and goals, the BDI architecture

treats intentions on a par with beliefs and goals, each defined in terms of accessible worlds. In the BDI architecture, beliefs, desires and intentions represent knowledge, motivation, and deliberative mental states of the agent respectively. To understand agents' rational behaviors, the interrelationships between beliefs, goals and intentions are formalized by introducing various axioms of change. The attitude of an agent (including future actions) is constrained by its present beliefs, goals, and intentions.

SRI international's PRS (Procedural Reasoning System) [73] is a framework based on the BDI model, described earlier in this section, for constructing real-time reasoning systems that can perform complex tasks in dynamic environments. Corresponding to the BDI model, PRS consists of (1) a database containing current beliefs and facts about the world, (2) a set of current goals to be achieved, (3) a set of plans, called *Acts*, describing how sequences of actions and tests may be performed to achieve certain goals or to react to particular situations, and (4) intentions containing those plans that have been chosen for execution. PRS attempts to achieve any goals it might have in light of its current beliefs about the world, while simultaneously reacting to any new events that occur. The dMARS (distributed Multi-Agent Reasoning System) [25, 26], a more recent version of PRS, reformalized those components in PRS to precisely specify the behavior of real PRS systems by defining the key data structures (including beliefs, goals, actions, plans and intentions) and the operations that manipulates these structures.

An intelligent agent architecture based on a Task Representation Language (TRL) has been developed at TAMU to provide realistic behaviors of aggregate-level units in battlefield simulations [44]. TRL allows specifying four fundamental types of knowledge: goals, tasks, methods and operators. Goals are conditions to be achieved; tasks are high-level actions to be done; methods are descriptions of specific procedures that can be used to accomplish tasks; and operators are primitive actions. TRL and PRS are similar in many aspects: 1) The knowledge base in TRL (architecture) is similar to PRS database for maintaining beliefs and facts about the world. 2) Both can specify goals to be achieved. 3) Tasks, and operators in TRL are similar to *Acts* in PRS for specifying actions to achieve certain goals; and methods in TRL is similar to *Act* plots in

PRS for specifying alternative procedures. 4) Intentions are the tasks (in TRL) and plans (in PRS) chosen to for execution. However, TRL and PRS have different emphases: TRL aimed at provide more expressivity to specify realistic behaviors of aggregate-level units while PRS focused on reasoning mechanisms so as to automate agents to achieve goals. PRS is more flexible to specify various types of goals (such as ACHIEVE, TEST and USE-RESOURCE) while TRL can only specify conditions as goals to be achieved. Goals are specified separately from *Acts* in PRS and the reasoning mechanism in PRS automatically chooses *Acts* to achieve goals. Considering that the reasoning mechanisms based on preconditions and effects may not be practical in battlefield domains under which tasks are hierarchical and complex, and that the methods for carrying out tasks in battlefield domains usually are predefined, goals in TRL are specified with tasks, and tasks and methods only have preconditions but no post-conditions or effects. While PRS represents *Acts* by a graph which allow specifying sequential, parallel and selective actions, TRL has richer expressivity to encode action process (such as WHILE, FOREACH and FORALL) and allows hierarchical tasks. In the TRL architecture, a mechanism of task decomposition is used to support the execution of hierarchical tasks. Considering the complexity of hierarchical tasks, tasks and methods in TRL include termination conditions to monitor their execution.

Researchers in Australian Artificial Intelligence Institute, including Rao, Sonenberg, Tidhar, Georgeff, and et al, developed a team-oriented programming language [84, 94, 107] (with different terms, including Team-Oriented Programming, Planned Team Activity and Social Plans), which semantics is formally defined on shared mental states, mutual beliefs, joint goals and joint intentions. For simplicity, we refer their team-oriented programming language as TOP. TOP is an implementation of BDI. In TOP, team activities are organized, in terms of roles, by a plan description language based on plan graphs. A plan graph contains START, AND, OR and END nodes representing synchronization. Social structures in TOP specify the levels of authority and responsibility that each team member possesses. To invoke a team plan, agents, guided by skills and capabilities, fill roles in the social structure of the plan [107, 108]. The

representation and execution of team plans are discussed in [64]. We note that our teamwork architecture, described in later chapters, also utilizes the concepts of role. Our concepts of roles are distinguished from the roles in TOP and other teamwork architectures discussed later in this section in these aspects: 1) while their delegation of roles to agents is made based the agents' capabilities, our delegation of roles to agents is made based on the social norms (i.e., constraints) on delegating the roles to agents, as well as the agents' capabilities; 2) our roles and role variables distinguish static (by roles) and dynamic (by role variables) action associations; and 3) when delegating roles and role variables in a plan to agents, we have the agents form a mental state (joint intention) to enforce the execution of the plan as a team effort, particularly the sub-actions in the plan will be executed coherently. The feature of (2) and (3) are not provided by roles in existing teamwork architectures.

The JACK Agent Architecture is a Java multi-agent environment based on the BDI model [10]. JACK Agent Language extends Java by introducing a small number of keywords for agent-oriented concepts (e.g., *agent*, *plan* and *event*) and statements for the declaration of attributes (e.g., the information contained in beliefs), the definition of static relationships (e.g., which plans can be adopted to react a certain event), and the manipulation of an agent's state (e.g., add new goals, change beliefs, and interact with other agents). JACK Teams™ extends JACK Agents to be a team-oriented framework by introducing the team reasoning entity, which encapsulates team behaviors in the same way that JACK Agent encapsulates agent behaviors. A role in JACK Teams is a distinct entity, which describes the relationship between participants in a team/sub-team, including the requirements of the role tenderer (i.e., the team requiring roles to be filled) and the role filler (i.e., the team providing performers to fill the roles), and sub-tasking specification (i.e., a set of roles calls a sub-task). A *teamplan* specifies how a task is achieved in terms of one or more roles. Team formation is based on the capability of a team, specified by *TeamCap*, which includes plans to handle initial team formation and default handling of a *TeamFormationEvent*, and the posting of a *StartTeamEvent*.

RETSINA [34, 100] (Reusable Environment for Task-Structured Intelligent Network Agents) is based on the assumption that all agents in teamwork have their own copy of a partially instantiated SharedPlan model of the task. In [34], the RETSINA model defines a role as a commitment by an individual, in the context of achieving an overall team goal, to (1) the accomplishment of a task, or to (2) the persistence of an activity until the reasons for performing that activity no longer hold, or until the individual is asked to decommit from that activity. When a team of agents pursues a task, each agent produces a set of candidate roles according to their capability to the task fulfillment requirements; and selects one at a time and proposes the selection to its teammates until all required capabilities of the task are covered. The agents commit to executing the task once all roles are selected and there are no role conflicts (i.e., a role is selected by more than one agents). Checkpoints representing expectations (i.e., committed agent has begun its role, is in the process of executing it, or has completed it) are used to confirm the team's global commitment to the team goal. The RETSINA MAS infrastructure [34] implements the RETSINA model to provide location service, ontologies, and language that allow agents to collaborate, exchange information and services. The agents in the RETSINA MAS infrastructure communicate with each other using a standard Agent Communication Languages (ACLs) [90].

STEAM (a Shell for TEAMwork) [103, 104] is a flexible and reusable teamwork architecture built on SOAR [56], in which states, goals and operators are similar to beliefs, desires and intentions in the BDI architecture and represent possible stages of progress, desired situation, and the transformation of states via some action [33]. STEAM allows creating a basic building block of joint intention [22, 59]. Teamwork in STEAM is realized by agents building up a (partial) hierarchy of joint intentions. Joint intentions can be specified by team plans and team operators. When agents perform team tasks, they play certain roles in the tasks according to their abilities. An interesting feature of STEAM is that team members can monitor the team's and individuals' performance and reorganize the team as necessary [101, 105]. When a team operator becomes unachievable because an agent loses the ability to perform a critical role, agents

synchronize to invoke a repairing procedure and inform teammate repair results. According to the repair result, another agent capable of performing the critical role takes over that role. Decision-theoretic communication selectivity reduces communication overhead of teamwork with appropriate sensitivity to the environmental conditions [80].

GRATE (Generic Rules and Agent Model Testbed Environment) [49, 53] agents have two components: a cooperative and control layer and a domain level system. The domain level system may be preexisting or built on purpose and solves problems in the domain. GRATE agents store all the domain-dependent information, which is necessary to define individual behaviors, in specific data structures called *agent models*. GRATE provides a set of constructs to represent the agent model knowledge, including state knowledge (e.g., solution progress), capability knowledge (e.g., task descriptions and recipes), intentional knowledge (e.g., intentions), evaluative knowledge (e.g., time to complete task and intention end time), and domain knowledge (e.g., recipe priority and recipe triggers). The cooperation and control layer is a meta-controller, which operates on the domain level system, in order to ensure that the activities on the domain are coordinated with those of others within the community. The knowledge in the cooperation and control layer is represented by generic rules written in standard if-then format. The cooperation and control layer was originally implemented by forward-chaining, and later by following the formalism of joint responsibility theory [47, 52], described in Sec. II.1.3.

A teamwork architecture, CAST (Collaborative Agents for Simulating Teamwork), has been developed by the MURI group at TAMU to simulate effective teamwork for general purposes [127, 128, 129]. In particular, CAST has been used to support agents in team training, under which a team is composed of software agents and human subjects. The knowledge in CAST is specified by a teamwork language, MALLET (Multi-Agent Logic Language for Encoding Teamwork) [28], which has rich expressivity to specify the mental states underlying teamwork and dynamic team structure. In CAST, shared mental models are used to coordinate agents' activities and further facilitate reasoning

mechanisms to improve team performance. More details about CAST and MALLET will be given later in the discussion of shared mental models (Sec. II.4).

II.3 Role

Biddle and Thomas [4] investigated the different interpretations of role in the role field and formed their role theory. In their role theory, they discussed the properties of role and the relationships among roles from behavioral, social, cognitive, normative and legislative perspective. They also gave classificatory concepts of role, which deal with a limited set of phenomenal referents, by using partitioning conceptual operations and a criterion for subclass operation. The referents applied to partitioning conceptual operation include behaviors, persons, and persons and their behaviors. Their classificatory concepts of roles capture various behavioral aspects of role. More details about their classificatory concepts of role will be given when we formalize position, role and role variable in Chapter IV. However, they did not give any computational formalization of role, particularly leading to computational mechanisms for multi-agent systems.

Even though a fundamental, comprehensive and computational formalization of role is still not available, many researchers still have been using the concept of role by making use of some properties of role or formalizing certain mechanisms by using the concept of role.

Roles have been used to organize team activities. Ferber et al. [29, 30] proposed a generic meta-model of organizations based on the concepts of roles, groups and organizations. The meta-model uses organizational reflection to describe the relationships between interaction and organization. Moreover, a formal specification was proposed to define role behavior and interaction requirements. In some teamwork architectures described earlier in Sec. II.2, including STEAM [103, 104], TOP, RETSINA, dMARS and JACK, team plans specify team activity in terms of roles. In these teamwork architectures, when a team of agents invokes a team plan to achieve a certain goal, the agents fill the roles in the team plan according to their skills and the capability requirements of the roles. In STEAM, agents can monitor and repair failures

through team operators and roles [101, 105]. In RETSINA, agents can dynamically drop roles and other agents can reselect the dropped roles as necessary [101].

Roles have been used to facilitate task decomposition and task delegation. Stone and Veloso [96] used roles to decompose tasks for Robocup soccer team. In [96], a role consists of the specification of an agent's internal behaviors (i.e., update the agent's internal state based on its current internal state, the world state, and the agreement among team) and external behaviors (i.e., the agent's interaction with the world). Roles are defined in terms of positions in the field of soccer game, such as midfielder. Team tasks then are decomposed to sub-tasks for the roles. Rather than being fixed to certain positions, robots coordinate and dynamically decide the team formation by role assignments. In [16], Castelfranchi and Falcone defined delegation and adoption based on the relationships between agents: "in delegation an agent A needs or likes an action of another agent B and includes it in its own plan"; and "in adoption an agent B has a goal since and until it is the goal of another agent A". The action in delegation and adoption is viewed as a task. Castelfranchi and Falcone viewed a role as a social relation regarding to a task. If agent A delegates task *t* to agent B (i.e., Delegate(A, B, t)), A is a role client and B is a role contractor. Then, a task delegation is a creation of a role. Further, they extended a role to be an abstract agent with goals and plans for those goals. The tasks in the plans are delegated through the role delegation.

Roles have been used to facilitate coordination among individuals. Barbuceanu et al. [1, 2, 3] have developed a coordination language, called COOL, which specifies coordination among individuals in terms of roles. At the organization level, roles specify social constraints on agent behaviors, in particular, the obligation of an agent filling a role to another agent with respect to performing actions to achieve a goal. At the individual agent level, obligations and interdictions were viewed as mental states much like beliefs, desires, and intentions. Being more specific, an obligation is defined as a generic rule, which specifies the condition under the obligation becomes active (:condition clause), the role being obligated (:obligated clause), the role that obligates (:authority clause), and the goal of the obligation (:goal clause). The control architecture

of COOL selects obligations and goals, and further coordination plans according to beliefs and generic obligation rules.

Roles have been used to form multi-agent theories to guide the construction of multi-agent architectures and systems. While the BDI model emphasizes practical reasoning based on beliefs, desires and intentions, Chainbi [18, 19] proposed a Belief-Goal-Role theory (BGR) to stress the concept of organization and thereby cooperation. Chainbi divided actions into four categories: physical actions (i.e., interactions between agents and the environment), communicative actions (i.e., interactions between agents), private actions (i.e., internal functions of agents), and decision actions (i.e., actions that generate communicative, physical and private actions). A role is defined as a sequence of actions. Chainbi gave a set of formalization about the condition under which an action of a role can be taken, and the belief changes of agents according to the category of the action. However, many issues remain unsolved, for example, how to specify a role, how to delegate roles to agents, how the delegation of a role to agent impacts on the agent's mental states. So his BGR is quite preliminary and no multi-agent architecture based on BGR has been seen.

II.4 Shared Mental Model

Johnson-Laird [54] defined mental models as working models by which human beings understand the world and interact with it. In [12], Cannon-Bowers et al. originally suggested that shared mental models, which extend the mental model to a context of a team, are "knowledge structures held by members of a team that enable them to form accurate explanations and expectations for the task, and in turn, to coordinate their actions and adapt their behavior to demands of the task and other team members". Cognitive studies [65, 85, 87] have shown that shared mental models can improve team performance and effective teams usually have some degree of shared knowledge. The term shared mental model is also named team mental model by some researchers [11, 57]. Researchers in various fields have developed other concepts, which are in distinct forms, but clearly related to shared mental models, including information sharing (e.g., pooling information by groups) [95], transactive memory (i.e., memory is a social

phenomenon and individuals often utilize each other as external memory aids to supplement their own limited and unreliable memories) [118], group learning (i.e., the construction of new knowledge by a group) [68], and cognitive consensus (i.e., the similarity among group members) [72].

Moreover, psychological researchers have shown that teams that maintain mutual awareness through shared mental models can improve the effectiveness with which they communicate and exchange information with each other, and assist each other [65, 85, 87].

Cannon-Bowers and Salas [11] laid out what shared mental models contain, including task-specific knowledge, task-related knowledge, knowledge of teammates and attitudes/beliefs. They also suggested that "shared" means overlapping, similar, identical, complementary, and/or distributed. In [57], Langan-Fox et al. pointed out that efficient shared mental models have high potential to make communication and coordination more efficient by requiring less communication between individuals for the same result, and to improve allocation of tasks and decision control through team awareness. In contrast, especially in a mixed team [69] of software agents and human subjects, inefficient shared mental models may cause software agents to produce a large amount of communication far beyond the capability of human subjects' handling communication.

The MURI group, across Texas A&M University, Pennsylvania State University and Wright State University, has been using intelligent agents in team training [69, 70], in which intelligent agents function in two ways: 1) intelligent agents play the roles, as virtual team members, together with human players, 2) intelligent agents monitor human players' behavior and tutor human players based on expert knowledge. Based on shared mental models, a teamwork architecture, CAST [127, 128, 129], has been developed to interpret teamwork knowledge, maintain the mental states underlying teamwork, coordinate agents' activities, and interact with application domains. The teamwork knowledge is represented by a teamwork programming language, MALLET [28]. MALLET can explicitly specify the mental states underlying teamwork (such as, mutual

beliefs, shared goal, joint intentions and team plans) and the knowledge of team structures (such as agents, agents' capabilities, teams).

In CAST [127, 128, 129], shared mental models, as a task-specific knowledge, are represented by an extension of Predicate/Transition (PrT) nets [122, 123], called CAST-PN. Each agent has a (partially) consistent copy of CAST-PN. Agents perform their teamwork by executing their CAST-PNs with coordination. Based on shared mental models, a mechanism of PIE (Proactive Information Exchange) has been developed to support information sharing among agents during performing their teamwork [124, 125, 126, 128]. In PIE, the potential information needs are inferred by reasoning about the goals of the other agents based on the team plan used by the agents. Being more specific, the preconditions of operators/plans to be performed by other agents pose information needs because the agent needs to know that information before the operators/plans are executed. Ioerger's PIEX [42] improved this mechanism by including reasoning about other agents' beliefs to reduce un-necessary information exchange. In PIEX, a variety of justifications are used to update beliefs about other agents' beliefs, including direct-observation, observability, effects of actions, inferences, persistence and assumptions. An agent proactively sends a message about some information only if the agent can infer that the other agent either does not believe the information or believes it incorrectly.

The main features distinguishing CAST and MALLET from other teamwork architectures are: 1) MALLET is more expressive than the agent languages used in other architectures. 2) CAST and MALLET are designed for general purposes. They can be used for different domains. The teamwork knowledge in different domains can be encoded in MALLET, and CAST can interpret the knowledge and drive agents to interact with the domain simulations. 3) CAST is built based on shared mental models, which maintains the mental states underlying teamwork and coordinate agents' activity. Through shared mental models, CAST can incorporate various reasoning mechanisms to improve team performance, such as proactive information exchange.

CHAPTER III
A TEAMWORK DESCRIPTION LANGUAGE - RoB-MALLET

III.1 Introduction

In this chapter, we will introduce RoB-MALLET (Role-Based Multi-Agent Logic Language for Encoding Teamwork). In our teamwork architecture, RoB-CAST (Role-Based Collaborative Agents for Simulating Teamwork), described in Chapter VI, the knowledge about teamwork is represented in RoB-MALLET.

RoB-MALLET is an extension of MALLET. RoB-MALLET extends MALLET mainly by introducing the concepts of position, role, and role variable to define plans and deprecating the constructs related to roles in MALLET, including RoleDef, PlaysRole, Fulfilledby, and agent variables. Though MALLET was developed as a team effort (of which the author was a part), until very recently[2], there has been only a meager, and in some ways incomplete, description of its semantics. There are critical issues of relating MALLET constructs to team mental states exploited by teamwork theories, such as mutual beliefs, shared goals, and joint intentions. Previous semantics and implementations of some constructs may diverge from mental states commonly accepted in the field of teamwork. In particular, distributed agents may not have consistent condition evaluations and thus not have consistent executions.

As we will discuss how RoB-MALLET includes the concepts of position, role and role variable in Chapter IV, we describe only the key concepts of RoB-MALLET and discuss the semantics of RoB-MALLET constructs in terms of mental states in this chapter. We will describe the RoB-MALLET syntax, which allows explicit specifying the mental states underlying teamwork. We will introduce both the RoB-MALLET constructs for individual knowledge, including individual operators, agents, capabilities, beliefs, goals and intentions, and the RoB-MALLET constructs for team knowledge,

[2] Recently, a paper by Fan, et al, has been completed and submitted for review.

including teams, mutual beliefs, shared goals[3], team operators/plans, JOINT DO's, and joint intentions. We will describe various RoB-MALLET constructs for flow control, such as sequential and parallel, branch (IF construct), loop (WHILE construct), FOREACH, and FORALL constructs. In particular, we will relate the evaluation of the conditions in these constructs to mutual beliefs so that agents have a globally consistent execution and further teamwork specified by the constructs is enforced.

While we give the RoB-MALLET syntax of various constructs when we elaborate these constructs, the complete RoB-MALLET syntax in BNF is given in APPENDIX A. Some constructs are used to express both individual and team knowledge and the syntax in APPENDIX A is merged for both; however, we split these constructs for individual and team knowledge in our description. For easier understanding, we may modify slightly the syntax from that in APPENDIX A to fit the flow of our description. However, what we present is logically equivalent to APPENDIX A.

One other important point to note is that the syntax for RoB-MALLET is centered around the use of positions, roles and role variables (as noted above), but as yet we have not given the formal definition of these terms. It is sufficient to note at this point that in execution, roles and role variables will be replaced by actual agents within the processes of role-based plans. Thus, our explanations of the semantics will described in terms like "the agent filling the role …". In Chapter IV, we will give a more formal and abstract definition of the terms position, role and role variable that enable us to appropriately discuss our views on responsibility and delegation, and then explain how these terms are given concrete representations from the syntax discussed here.

In the RoB-MALLET BNF, "<IDENTIFIER>" represents a string of integers and letters starting with a letter; "<VARIABLE>" represents a string of integers and letters starting with "?"; "<INTEGER>" represents a string of integers; reserve words are not case-sensitive but we write them in upper case to distinguish them from other types of identifiers; "<NOT>", "<EQ>", "<LT>", "<GT>", "<LE>", and "<GE>" are used in the

[3] In this dissertation, belief, goal, and intention mean individual belief, individual goal, and individual intention respectively.

syntax of predicate and mean negation, equal to, less than, greater than, less than or equal to, and greater than or equal to respectively; "|" means options, i.e., either one in the list; "*identifier*?" means that *identifier* is optional; "*identifier**" means that *identifier* is optional or repeatable at any number of times; and "*identifier*+" means that *identifier* is repeatable at any number of times but at least once.

III.2 Individual Agent Knowledge Specification in RoB-MALLET

Individual agent knowledge specification in RoB-MALLET includes the specifications of agents and capabilities, beliefs, goals, individual operators, and intentions,

III.2.1 Agent

Agents in RoB-MALLET are defined using the following syntax:

AgentDef ::= "(" "AGENT" AgentName ")"

AgentName ::= <IDENTIFIER>

For example, `(AGENT ag1)` declares ag1 as an agent.

III.2.2 Belief

In RoB-MALLET, agents' beliefs are represented as predicates in the infix format of nested lists. The following is the syntax of predicate in RoB-MALLET:

Pred ::= "(" (<IDENTIFIER> | <NOT> | <EQ> | <LT> | <GT> | <LE> | <GE>)
(<IDENTIFIER> | <VARIABLE> | Pred) ")"*

RoB-MALLET provides constructs *ASSERT* and *RETRACT* to update beliefs. The *ASSERT* construct is to add new beliefs while the *RETRACT* construct is to remove existing beliefs. To update a belief, agent can retract existing belief and assert new belief. The syntax *ASSERT* and *RETRACT* constructs is as follows:

AssertDef ::= "(""ASSERT" Pred+ ")"

RetractDef ::= "(" "RETRACT" Pred+ ")"

RoB-MALLET also provides a construct to specify rules on belief reasoning. Rules in RoB-MALLET are based on Horn clauses [86] and a rule contains a list of predicates.

If a rule only contains a predicate, it becomes a belief (fact); otherwise, the first predicate is true if all other predicates are true. The following is the syntax of rule in RoB-MALLET:

RuleDecl ::= "(" "RULE" Pred+ ")"

For example, `(dog fido)` means that fido is a dog; and `(NOT (dog fido))` means that fido is not a dog. `(animal fido)` means that fido is an animal. `(ASSERT (dog fido))` means that an agent adds a belief that fido is a dog. `(RETRACT (dog fido))` mean an agent removes a belief that fido is a dog. `(RULE (animal ?x) (dog ?x))` is a rule which means if ?x is a dog then ?x is an animal. Suppose an agent has a belief that `(dog fido)`, the agent can infer another belief `(animal fido)` according the rule.

In addition, actions (including operators and plans) can update agents' beliefs. We will discuss it when we discuss them later in Sec. III.2.4 and III.3.4.

III.2.3 Goal

A goal of an agent in RoB-MALLET is a predicate or a conjunction of predicates, which is currently false and the agent desires to be true in the future. The following is the syntax of a goal of an agent in RoB-MALLET:

GoalDef ::= "(" "GOAL" AgentName Pred+ ")"

In later sections, we will explain how RoB-MALLET explicitly expresses shared goals of a team of agents. The syntax of shared goals is similar, just replacing *AgentName* with *TeamName*. However, they are semantically different and have different impacts on the mental models of agents. A declaration of a goal of an agent is to add an individual goal to the mental model of the agent. For example, `(GOAL ag1 (happy ag1))` means that agent ag1 has a goal, ag1 is happy, in its mental model.

III.2.4 Individual Operators

The concept of action has been formalized as state transition [31, 62] and used in logic programming languages, such as an operator with a precondition and a post-

condition (represented by a delete list and an add list) in STRIPS [61]. An action in RoB-MALLET essentially is a syntactic variation of a STRIPS action (operator or plan). In RoB-MALLET, an action is generally specified by action name, a set of preconditions, and a set of effects. The execution of an action means, the action occurs when current state satisfies the preconditions (e.g., the preconditions are among the agent's beliefs) and results in a state in which the effects are true (e.g., the effects become agent's beliefs).

In RoB-MALLET, actions are defined in terms of operators (including individual and team operators) and plans (could be either individual or team plans). Individual operators (iopers) are atomic actions that may be taken in the environment. The following is the syntax of individual operator:

IOperDef ::= "(" "IOPER" OperName VariableListOpt PreConditionList? EffectsList?)

OperName ::= <IDENTIFIER>

VariableListOpt ::= "(" <VARIABLE> ")"*

PreConditionList ::= "(" "PRE-COND" Pred+ (":IF-FALSE" ("FAIL" | "WAIT" (<INTEGER> | <VARIABLE>)? | "ACHIEVE"))? ")"

EffectsList ::= "(" "EFFECT" Pred+ ")"

The preconditions of individual operators (iopers) are conjunctions of predicates, in which variables are allowed. When an agent is supposed to perform an ioper, the ioper is actually performed only if all predicates in its preconditions are true. There are three modes of handling false preconditions, including *fail*, *wait* and *achieve* modes. If the preconditions are false, the ioper is not executed in the *fail* mode; the agent in the *wait* mode keeps the checking of the preconditions until they become true and then the ioper is executed, or if the timeout is reached without the precondition becoming true, the ioper is not executed. In the *achieve* mode, the agent should find a way to achieve the preconditions, after which the ioper occurs; if the agent cannot achieve the preconditions, then the ioper is not executed.

The effects of iopers are conjunctions of predicates, in which variables are allowed. The occurrence of the ioper results in a state in which all predicates in its effects are true.

Variables may be used in the predicates of preconditions and effects. If any of them is in the *VariableListOpt* of the operator, the actual value of the variable, which is passed in when the operator is called, substitutes the variable in the predicates. If a variable contained in a predicate is unbound with any actual value, then a fact with any value for the variable can satisfy the predicate.

For example, `(IOPER buy (?goods, ?price) (PRE-COND (GT money ?price) :IF-FALSE FAIL) (EFFECTS (have ?goods)))` declares an ioper `buy` with variables `?goods` and `?price`. The precondition is that money is more than `?price`; and the effect is that the agent has `?goods`. If the precondition is not true, ioper `buy` fails to occur. If operator buy is called with arguments (DVD, 100), ?goods and ?price are bound with values DVD and 100 respectively.

In RoB-MALLET, another type of individual action is the individual plans. We treat an individual plan as a team plan with only one member in the team. Therefore, there is no explicit construct for individual plans in the RoB-MALLET. We will give the syntax and semantics of plan later when we discuss the team knowledge specification in RoB-MALLET (Sec. III.3.4.3).

III.2.5 Individual Intention

In RoB-MALLET, an intention can be specified in terms of either a goal or an action. The following is the syntax of individual intention in RoB-MALLET:

Achieve ::= "(" "ACHIEVE" RoleName Pred+ ")"

Start ::= "(" "START" AgentName Invocation ")"

Invocation ::= "(" PlanOrOperName (<IDENTIFIER> | <VARIABLE>) ")"*

DoMalletProcess ::= "(" "DO" (RoleName|RoleVariableName) MalletProcess ")"

Achieve defines individual intentions in term of goals, which are sets of predicates. *Start* defines individual intentions in term of actions, which are individual operators or plans. *DoMalletProcess* is used in the processes of plans to define an individual

intention. It means that, the agent filling the role (or role variable) intends to execute the action in the *MalletProcess*. We will describe plans and processes later in Sec. III.3.4.3. For example, `(ACHIEVE ag1 (happy ag1))` means that agent ag1 intends to achieve a goal, `(happy ag1)`. `(START ag1 (catch deer))` means that ag1 intends to take action catch (could be either operator or plan).

The semantics of individual intentions in RoB-MALLET are based on Cohen and Levesque's persistent goal [22, 59]. An agent has a persistent goal relative to *q* to achieve *p* iff

1. the agent believes that *p* currently false;
2. the agent wants *p* to be true eventually;
3. condition 2 will continue to hold until the agent believes that *p* is true, or that *p* will never be true, or that *q* fails.

Agent *ag*'s intention to achieve goal *g* (i.e., `(ACHIEVE` *ag g*`)` in RoB-MALLET) means that agent *ag* has a persistent goal of having achieved *g* and, moreover, having achieved *g* believing throughout that the agent is achieving *g*. Agent *ag*'s intention to execute action *a* (i.e., `(START` *ag* `(a))` in RoB-MALLET) means that, agent *ag* has a persistent goal of having done action *a* and, moreover, having done action *a* believing throughout that the agent is doing action *a*. We note that the "*q*" in the persistent goals is true except for an intention to execute a plan (i.e. action *a* is a plan). For an intention to execute a plan, the "*q*" is the termination condition of the plan.

III.2.6 Capability

Capability is a broad concept and many factors can relevant to its definition. Capability can be formalized as "know-how" concerning agents' mental states [40]. Capability also can be constrained by available resource and workload (e.g., whether an agent is busy and how busy) [38].

For our purpose, the capability of an agent in RoB-MALLET is defined in terms of a set of operators. If the capability of an agent *ag* is a set of operators op_1, op_2, …, op_n, the

agent *ag* is capable to perform these operators. The following is the syntax of capability in RoB-MALLET:

CapabilityDef ::= "(" "CAPABILITY" (AgentName | "("AgentName+ ")") "(" OperName+ ")" ")"

For example, `(CAPABILITY ag1 (play study sleep))` declares that agent ag1 is capable to do operators `play`, `study`, and `sleep`.

Although plans also include actions as operators, we do not define capability in term of plans. The reason for not doing so is that we take Grosz's [36] view of complex actions that complex actions must eventually rest in individual actions of agents but may be more complex than the aggregate of individual actions. Therefore, plans, as complex actions, must eventually rest in individual operators.

III.3 Team Knowledge Specification in RoB-MALLET

One of main features distinguishing RoB-MALLET from other agent languages, such as SOAR, PRS, and COOL, is that RoB-MALLET provides constructs to explicitly specify team knowledge, including team structure and team mental states (such as mutual belief, shared goals, team operators, JOINT DO's, team plans, and joint intentions). In this section, we will elaborate how RoB-MALLET specifies team knowledge in syntax and how team knowledge is different from its corresponding individual agent knowledge in semantics, for example, shared goal versus individual goal.

III.3.1 Team

Teams in RoB-MALLET are defined using the following syntax:

TeamDef ::= "(" "TEAM" TeamName "("AgentName+ ")" ")"

TeamName::= <IDENTIFIER>

For example, `(TEAM team1 (ag1 ag2))` declares agent ag1 and ag2 as team team1.

Teams in RoB-MALLET are heterogeneous. There may be more than one team. Different teams are formed for different reasons. For example, agents may form a team

for a shared goal to be achieved, or agents may form a team for a joint intention to do team operators or plans. Different teams may contain different numbers of different agents and different agents may have different capabilities (defined in Sec. III.2.6). Two teams may overlap, i.e., some of agents in one team may also be members of another. An agent may be a member of multiple teams.

III.3.2 Mutual Belief

We first note that a mutual belief is more than the aggregate of individual beliefs. If a team of agents has a mutual belief φ, every agent in the team has not only an individual belief φ, but also an individual belief that other agents in the team believe φ. A mutual belief φ of team T is defined in an infinite recursive fashion in [27] as follows:

- Every agent in T believes φ, this is denoted by $E_0(T, \varphi)$;
- Every agent in T believes $E_0(T, \varphi)$, and this is denoted by $E_1(T, \varphi)$. Also, Every agent in T believes $E_1(T, \varphi)$, and this is denoted by $E_2(T, \varphi)$. Morever, every agent has such beliefs on $E_k(T, \varphi)$ infinitely and recursively.
- The mutual belief is defined as the infinite conjuction of $E_i(T, \varphi)$ (where i starts from 0), denoted by $E(T, \varphi)$.

Instead to represent such infinitely recursive beliefs, we represent an approximation of mutual belief. That is, every agent in T believes φ (i.e., $E_0(T, \varphi)$), and every agent believes that every agents believes $E_0(T, \varphi)$ (i.e., $E_1(T, \varphi)$). To represent such an approximation of mutual belief, every agent in T needs to represent not only its belief on φ (i.e., $E_0(T, \varphi)$) but also its belief on also other agents' beliefs on $E_0(T, \varphi)$ (i.e., $E_1(T, \varphi)$). However, rather than represent mutual belief explicitly in each agent's mental model, we evaluate whether or not mutual belief holds whenever a condition involving mutual belief is evaluated. If we were to include a representation of mutual belief for each agent, the maintenance of the mutual belief would have significantly more overhead and we would have to complicate the semantics of individual belief. For example, suppose agents *ag1* and *ag2* have a mutual belief `(dog fido)`. If agent *ag2* were to retract its belief `(dog fido)`, then agents *ag1* and *ag2* would no longer have a

mutual belief `(dog fido)`. If we include the maintenance of the beliefs about other agents' beliefs in the semantics of mutual belief, we would then need to modify the semantics of individual belief. That is, when an agent retracts an individual belief, the agent would need to change other agents' beliefs about whether the agent has such belief.

Thus, rather than focus on the maintenance of the beliefs about other agents' beliefs for mutual beliefs, we concentrate on the evaluation of mutual belief. Strictly speaking, we do not represent mutual belief because we do not represent what one agent believes another agent believes. However, in a pragmatic sense, the mutual belief is only important at the points in execution when conditions involving mutual belief are evaluated. As per the semantics we will give below, when a condition calling for mutual belief is evaluated, we check the individual beliefs of all agents regarding the condition (i.e., evaluate $E_0(T, \varphi)$) and then inform all agents about whether or not all agents believe the condition (i.e., inform all agents of the evaluation of E_0) (T, φ). We impose additional meanings on "inform all agents of the evaluation of $E_0(T, \varphi)$": 1) if an agent informs other agents of the evaluation of E_0, the agent believes not only the evaluation of $E_0(T, \varphi)$ but also that the other agents will believe the evaluation of $E_0(T, \varphi)$; and 2) if an agent receives the evaluation of $E_0(T, \varphi)$, the agent believes not only the evaluation of $E_0(T, \varphi)$ but also that the sender and other receivers believe the evaluation of $E_0(T, \varphi)$. If agents have different beliefs on φ, then the evaluation of $E_0(T, \varphi)$ is false, moreover, all agents are so informed of the evaluation and thus the agents do not have a mutual belief on φ.

There are two methods for implementing the evaluation of $E_0(T, \varphi)$: 1) an involved agent serves as coordinator and evaluates the condition; and 2) all agents evaluate the condition separately. In first method, an involved agent serves as a coordinator to collect other agents' individual beliefs, evaluate the condition based on all involved agents' individual beliefs, and then propagate the result to all other involved agents. In second method, each agent collects other agents' individual beliefs and evaluates the condition based on all involved agents' individual beliefs. Although these two methods can make

agents have a consistent result, the first method is more efficient in communication as it just requires one agent collect other agents' beliefs and later propagate the result of the evaluation to other agents. In Chapter IV, we take the first method to handle the evaluation of a condition as a mutual belief when a team responsibility is decomposed into individual responsibilities, i.e., let an individual (a role or a role variable) serve as a coordinator to evaluate the conditions.

To represent a mutual belief, all involved agents maintain individual beliefs correspondingly. Strictly speaking, such a representation is not mutual belief. We call it psudo representation of mutual belief. However, with the semantics described above, the execution semantics achieve mutual belief in essence.

RoB-MALLET provides constructs *ASSERT* and *RETRACT* to update mutual beliefs[4]. *ASSERT* construct is to add new mutual beliefs. To assert a new mutual belief, each agent asserts the corresponding individual belief (i.e., the predicates of the mutual beliefs). *RETRACT* construct is to remove existing beliefs. To retract a mutual belief, each agent removes the corresponding individual belief (i.e., the predicates of the mutual beliefs). To update a belief, agent can retract existing mutual belief and assert new mutual belief. The syntax of *ASSERT* and *RETRACT* constructs is as follows:

AssertDef ::= "(""ASSERT" (TeamName | "("AgentName+ ")") Pred+ ")"

RetractDef ::= "(" "RETRACT" (TeamName | "("AgentName+ ")") Pred+ ")"

AssertMalletProcess ::= "(""ASSERT" "(" (RoleName | RoleVariableName)+ ")" Pred+ ")"

RetractMalletProcess ::= "(" "RETRACT" "(" (RoleName | RoleVariableName)+ ")" Pred+ ")"

In the above syntax, there are two sets of *ASSERT* and *RETRACT* constructs, *AssertDef* and *RetractDef*, and *AssertMalletProcess* and *RetractMalletProcess*. The first set is directly used to update agents' mutual beliefs. The second set is used in the

[4] The ASSERT and RETRACT constructs for updating mutual beliefs are available in RoB-MALLET, but not MALLET.

processes of role-based plans (described later in Chapter IV) to update the mutual beliefs of the agents which fill the roles (or role variables).

For example, team `T1` contains agents *ag1* and *ag2*. `(ASSERT T1 (dog fido))` means that both *ag1* and *ag2* add an individual belief that fido is a dog. `(RETRACT T1 (dog fido))` mean that both *ag1* and *ag2* remove an individual belief that fido is a dog.

In addition, joint actions (including team operators and plans) can update agents' beliefs. We will discuss it when we discuss them later in III.3.4.

The semantics of the evaluation of mutual beliefs are critical for RoB-MALLET and further RoB-CAST, particularly for leading to consistent execution among agents. In later sections, the evaluations of preconditions of team operators and plans (in Sec. III.3.4) and those of conditions in flow controls (in Sec. III.4) are based on mutual beliefs of involved agents.

III.3.3 Shared Goal

A team of agents may have a common goal, called shared goal in RoB-MALLET. A shared goal is a predicate or a conjunction of predicates, which is currently false and desired by the agents to be true in the future. The following is the syntax of a goal of an agent in RoB-MALLET:

GoalDef ::= "(" "GOAL" TeamName Pred+ ")"

GoalMalletProcess ::= "(" "ACHIEVE" "("RoleName+ ")" Pred+ ")"

In the above syntax, the first is directly used to set a shared goal of a team of agents. The second is used in the processes of role-based plans (described later in Chapter IV) to set a shared goal of the agents filling the roles in *GoalMalletProcess*.

In contrast to individual goals, a shared goal *g* of a team *t* means, all agents in team *t* not only believe *g* is a goal but also mutually believe all agents have *g* as a goal. For example, suppose a team team1 is declared as `(TEAM team1 (ag1 ag2))`, `(GOAL team1 (happy team1))` means that agent ag1 and ag2 have a goal `(happy`

`team1)` and that both ag1 and ag2 mutually believe that another has a goal `(happy team1)`.

III.3.4 Team Operator, Joint Do and Team Plan

Besides individual operators, actions in RoB-MALLET also include team operators, JOINT DO's and plans. Unlike individual operators, team operators, JOINT DO's and plans are collaborative actions, which eventually rest in individual actions of participating agents but coordination is required among these agents. In RoB-MALLET, the definition of team operators, JOINT DO's and team plans allows one to explicitly specify such coordination.

III.3.4.1 Team Operator

Analogous to individual operators, the specification of team operators also includes team operator names, preconditions and effects. Moreover, team operators are collaborative actions and coordination among agents can be specified. If an involved agent participates in the execution of a team operator according to the coordination of the team operator, the agent does so together with other agents at the same time. The following is the syntax of team operator in RoB-MALLET:

TOperDef ::= "(" "TOPER" OperName VariableListOpt (AND | OR | XOR)? PreConditionList? EffectsList? ")"

The preconditions of topers (team operator) are conjunctions of predicates, in which variables are allowed. When a toper is undertaken by a team of agents, it occurs if all agents mutually believe that all predicates in its preconditions are true. In other words, the preconditions of topers are evaluated as the mutual beliefs of involved agents. The preconditions are satisfied only if all involved agents have the corresponding beliefs. There are also three modes of handling false preconditions, including fail, wait and achieve modes. Handling false preconditions of team operators is similar to that of individual operators, except that a team operator requires participant agents mutually, not individually as individual operators, believe that all predicates in its preconditions are true.

The effects of topers (team operator) are conjunctions of predicates, in which variables are allowed. After a toper occurs, all participating agents mutually believe that all predicates in its effects are true.

We note that our semantics of preconditions and effects of team operator is consistent with teamwork theories. Our semantics can guarantee that all involved agents execute a team operator or none of them execute the team operator. That is essentially important for teamwork. For example, Cohen and Levesque's joint intention is to ensure that all involved agents perform a joint action, or that all involved agents mutually believe that the joint action cannot be done. As we describe later in Sec. III.3.5, an invocation of a team operator is a joint intention. In current MALLET/CAST, every involved agent just evaluates the preconditions separately based on its individual beliefs. So, they may have different results and consequently different decision on whether the team operator is executed. For example, suppose a team of agents invokes a team action (e.g., team operator or plan) with preconditions. Individual evaluation of preconditions just makes the execution of the team action look like an aggregation of the executions of individual actions. Indeed, some might try to execute it while others would not. That is, agents cannot form joint intentions to execute team actions; further, we can say that the agents would just separately run their individual plans and that occasionally the individual plans are the same. Apparently, such semantics and implementation are not consistent with teamwork theories. Although one may have alternate semantics of preconditions, using the semantics consistent with existing teamwork theories can help our RoB-MALLET to be accepted by the research community of teamwork, and can enable us to follow teamwork theories to build our teamwork architectures and build various reasoning mechanisms. This analysis also can be applied to the semantics of preconditions, effects and termination conditions of plans (later in Sec. III.3.4.3) and conditions in the constructs of flow controls (such as IF construct and WHILE construct) (later in Sec. III.4).

There are three modes of coordination, AND, OR, and XOR, among participating agents to execute team operators. The semantics of these three modes are defined in the following:

- AND requires all participating agents simultaneously execute corresponding individual operator;
- OR requires at least one participating agent to execute the corresponding individual operator. And if more than one agent participate, the agents execute the corresponding operator concurrently;
- XOR requires exactly one agent execute corresponding individual operator without conflict.

For example, suppose wumpus ?w is too big for a fighter to kill by self, but it can be killed by two fighters. A team operator for killing ?w may be written as:

```
(TOPER kill (?w)   AND
   (PRE-COND (alive ?w))
   (EFFECT (dead ?w))
)
```

That a team T1 invokes team operator `kill` with argument w1 is represented in RoB-MALLET by `(DO T1 (kill w1))`.

III.3.4.2 Joint Do

As one distinguishing feature of RoB-MALLET, team operators allow one to explicitly specify collaborative behaviors. As explained in last section, team operators require coordination on evaluating and handling preconditions, executing individual operators, and applying effects. The preconditions, effects, and individual operators are symmetric to all participating agents, i.e., all agents check exactly same preconditions, execute exactly same individual operators, and apply exactly same effects.

However, many scenarios in reality require as strong coordination as team operators, but they do not require so strict symmetries as team operators. For example, ag1 takes a picture of ag2. In this scenario, agents ag1 and ag2 are asymmetric in three aspects:

- The individual actions of agents ag1 and ag2 are different. Agent ag2 does operator smile while agent ag1 does operator take-picture.
- The preconditions of the operators of agents ag1 and ag2 are different. The precondition of agent ag1's operator take-picture is that agent ag1 beileves that ag2 is smiling. The precondition of agent ag2's operator smile is that agent ag2 believes that agent ag1 has set his camera ready.
- The effects of the operators of agents ag1 and ag2 are different. The effect of agent ag1's operator take-picture is that a picture of ag2 is made. The effect of agent ag2's operator smile is null (i.e., operator smile has no effect).

However, there is strong coordination between agents ag1 and ag2. Both ag1 and ag2 need to perform their operators, and at the same time.

To express such asymmetric team actions with strong coordination, RoB-MALLET extends team operator to JOINTDO. The following is the syntax of JOINTDO in RoB-MALLET:

JoinDoMalletProcess ::= "(" "JOINTDO" (AND | OR | XOR)? MalletProcess + ")"

The following is the RoB-MALLET code, using JOINTDO construct, for the action that the agent ag1 filling role r1 takes picture of the agent ag2 filling role r2:

```
(IOPER shot (?x)
   (PRECOND (smiling ?x))
)
(IOPER smile ()
   (PRECOND (ready camera))
)
(JOINTDO AND
   (DO r1 (shot r2))
   (DO r2 (smile))
)
```

As do team operators, JOINTDO has three modes of coordination, AND, OR, and XOR, among participating agents. The semantics of these three modes are exactly same as those in team operators, except coordinating different individual operators.

III.3.4.3 Team Plans

As well as team operators and JOINT DO's, team plans are complex actions. Similar to team operators, a team plan is identified by a plan name; and its execution changes the environment from a state, under which a set of conditions (i.e., the precondition of the plan) is satisfied, to another state, under which a set of conditions (i.e., the effects of the plan) is satisfied. In contrast to team operators, though, a team plan organizes agents' actions by a process, Moreover, a team plan may include a set of termination conditions, which agents keep monitoring during the execution of the team plan. The following is the syntax of team plans in RoB-MALLET:

PlanDef ::= "(" "PLAN" PlanName VariableListOpt RoleTeamDef ConstraintsList? PreConditionList? EffectsList? TermConditionsList? "(" "PROCESS" MalletProcess ")" ")"

TermConditionsList ::= "(""TERMCOND" ("SUCCESS"|"FAILURE")? Pred+ ")"

When a team plan is invoked by a team of agents, it occurs if all agents mutually believe that the preconditions of the team plan are true. There are also three modes of handling false preconditions, including *fail*, *wait* and *achieve* modes. Handling false preconditions of team plans is similar to that of team operators. After the team plan has been executed, all agents mutually believe that the effects of the team plan are true.

The process of a plan is similar to a function or procedure in other programming languages. It consists of invocations of individual operators, team operators, JOINT DO's and sub-plans (i.e., some agents do a sub-plan in the process) constrained by arbitrary combinations of flow controls (described later in Sec. III.4), such as sequential, parallel, branch and loop. If the process contains ACHIEVE statements for agents to achieve sub-goals, the team plan is an incomplete plan, similar to Grosz and Kraus' partial shared plan; otherwise, the plan is a complete plan, similar to Grosz and Kraus'

full shared plan. If the actions in the process are only performed by a single agent, the plan is an individual plan. In RoB-MALLET, individual plans are treated as a special type of team plans. When agents execute a team plan, the agents execute the actions in the process of the team plan by following the flow controls on the actions. If an action is a complex action, i.e., an invocation of a team operator, JOINT DO or sub-plan, the involved agents execute the action by following its corresponding semantics.

The termination conditions in a team plan are used to enhance the execution of the team plan. During the execution of the plan, the participating agents keep monitoring its termination conditions. If the termination conditions become true, the agents stop the execution of the plan. The termination conditions of a team plan essentially embody Cohen and Levesque's idea about "relative to *q*" in their joint intention theory [22, 59] (described in Chapter II) and Jennings' convention [50] (described in Chapter II). . To ensure that the execution of a team plan is a joint intention of the involved agent, the semantics of termination conditions should guarantee that all involved agents have the same attitude to the execution of the plan, either that all agents do not stop the execution, or that all agents stop the execution together. Two optional semantics can be applied to termination conditions: 1), the termination conditions are evaluated based on the mutual beliefs of the agents so that all agents have the same decision on whether or not to stop the execution of the team plan; or 2), each agents evaluate the termination condition based on the individual beliefs. If an agent found that the termination conditions become true, the agent coordinate all other agents to stop the execution together. To be consistent with the semantics of preconditions and effects, we use the first semantics in this dissertation.

We would like to note that, our semantics of preconditions, effects and termination conditions of team plans, which is based on the mutual beliefs of the involved agents, also guarantees that the execution of team plans are joint intentions of the involved agents. All agents start the execution of team plans together because they have a consistent view on the preconditions. All agents have the same decision on whether to stop the execution and stop the execution of team plans together because they have a

consistent view on the termination conditions. In current MALLET/CAST, every involved agent just evaluates preconditions and termination conditions based on its individual beliefs. It could happen that some agents start the execution of team plans but other agents do not. It also could happen that some agents stop the execution of team plans but other agents still continue to execute the team plans.

RoleTeamDef and *ConstraintsList* are related to our notion of role-based plan described in Chapter IV. Their syntax and semantics will be elaborated in Chapter IV.

The following is an example of a team plan, `killwumpus(?w)`. The precondition of this plan is that `?w` is alive. To kill wumpus `?w`, the agent filling role r1 executes operator `findwumpus` to find `?w`, the agent filling role r2 or r3 executes operator `movetowumpus` to move close to `?w`, and then executes operator `shootwumpus` to shoot `?w`. The effect of this plan is that `?w` is dead. However, the agent filling role r1 and the agent filling role r2 (or r3) stop executing this plan if `?w` hides.

```
(PLAN killwumpus(?w)
   (ROLETEAM ((role (r1) sniffer) (role (r2 r3)
              fighter)))
   (PRECOND (alive ?w))
   (EFFECT (dead ?w))
   (TERMCOND (hide ?w))
   (PROCESS
         (SEQ
               (DO r1 (findwumpus ?w))
               (Select ?fi (r2 r3) (closestto ?fi ?w))
               (DO ?fi (movetowumpus ?w))
               (DO ?fi (shootwumpus ?w))
         )
   )
)
```

We note that ?fi is a role variable we will explain later in Chapter VI. Basically, a role variable is for dnynamically associating actions to some role. In the above example, the one which of role r2 and r3 is closer to the wumpus ?w is selected to execute operators `movetowumpus` and `shootwumpus`.

III.3.5 Joint Intention

In RoB-MALLET, a joint intention can be specified in terms of either a shared goal or a team action (team operator or plan). The following is the syntax of joint intention in RoB-MALLET:

Achieve ::= "(" "ACHIEVE" TeamName Pred+ ")"

Start ::= "(" "START" TeamName Invocation ")"

Invocation ::= "(" PlanOrOperName (<IDENTIFIER> | <VARIABLE>) ")"*

DoMalletProcess ::= "(" "DO" ByWhomSpec MalletProcess ")"

ByWhomSpec ::= RoleName | RoleVariableName | "(" (RoleName | RoleVariableName)+ ")"

The above syntax is similar to that of individual intention in Sec. III.2.5, except that *AgentName* is replaced with *TeamName*. *Achieve* defines joint intentions in term of goals, which are sets of predicates. *Start* defines joint intentions in term of actions, which are team operators or plans. The above *DoMalletProcess* also defines joint intentions in terms of actions. It means, the agents filling the roles (or role variables) intend to execute the action in the *DoMalletProcess*. For example, `(ACHIEVE t1 (happy all))` means that team t1 jointly intends to achieve a goal, `(happy all)`. `(START t1 (killwumpus w1))` means that team t1 jointly intends to execute plan killwumpus.

The semantics of joint intention in RoB-MALLET are based on Cohen and Levesque's joint persistent goal [22, 59]. A team of agents has a persistent goal relative to *q* to achieve *p* iff

1. they mutually believe that *p* currently false;
2. they mutually know they all wants *p* to be true eventually;

3. condition 2 will continue to hold until they mutually believe that *p* is true, or that *p* will never be true, or that *q* is fails.

The joint intention of agents in team T1 to goal *g* (i.e., `(Achieve T1 g)` in RoB-MALLET) means that, the agents in team T1 have a joint persistent goal of having achieved *g* and, moreover, having achieved *g* mutually believing throughout that the agents are achieving *g*. The joint intention of agents in team T1 to execute action *a* (i.e., `(START T1 (a))` in RoB-MALLET) means that, the agents in team T1 have a joint persistent goal of having done action *a* and, moreover, having done action *a* mutually believing throughout that the agents are doing action *a*. We note that the "*q*" in the joint persistent goals is true except for a joint intention to execute a plan (i.e., action *a* is a plan). For an intention to execute a plan, the "*q*" is the termination condition of the plan.

As explained in Section III.3.4.3, a plan contains a process of actions. The actions in the process could be either individual actions, which appear as *(DO RoleName MalletProcess)* or *(DO RoleName MalletProcess)* (defined by the syntax of *DoMalletProcess* in Sec. III.2.5), or joint actions, which appear as *(DO RoleAndRoleVariableList MalletProcess)* (defined by the syntax of *DoMalletProcess* above in this section). The complete BNF definition of *DoMalletProcess* is given in APPENDIX A. As explained earlier, an invocation of a plan is a joint intention. An individual action in the process of the plan is a sub-individual-intention of the joint intention. A joint action in the process of the plan is a sub-joint-intention of the joint intention.

III.4 Flow Controls

In RoB-MALLET, various constructs of flow control are used to organize actions in the processes of team plans. Different flow controls pose different temporal constraints and coordination between the actions in the flow controls. Basic types of flow controls include sequential, parallel, branch, and loop. In this section, we will describe the syntax and semantics of these flow controls. The definition of *MalletProcess* is given in APPENDIX A.

III.4.1 Sequential

The following is the syntax of sequential flow control in RoB-MALLET:

SeqMalletProcess ::= "(" "SEQ" MalletProcess+ ")"

A *SeqMalletProcess* may consist of arbitrary (but at least one) number of *MalletProcess*es. These *MalletProcess*es must be executed in the order they appear. If multiple agents are involved, they must coordinate as necessary to guarantee the order. For example,

```
(SEQ
    (DO r1 (op1))
    (DO r2 (op2))
)
```

The agent ag1 filling r1 and the agent ag2 filling r2 execute the above sequential process. They must coordinate and guarantee that ag1 executes op2 after ag1 finishes op1.

III.4.2 Parallel

The following is the syntax of parallel flow control in RoB-MALLET:

ParMalletProcess ::= "(" "PAR" MalletProcess+ ")"

A *ParMalletProcess* may consist of arbitrary (but at least one) number of *MalletProcess*es. These *MalletProcess*es are executed concurrently. For example, suppose there is a PAR statement as the follows:

```
(PAR
    (DO r1 (op1))
    (DO r2 (op2))
)
```

The agent ag1 filling role r1 and the agent ag2 filling role r2 can execute `op1` and `op2` in parallel, or in any relative order, in the following parallel process. However, if this *par* is within a *seq*, neither agent may proceed past the end of the *par* until both are ready to do so. Similarly, neither can enter it until both are ready to do so.

III.4.3 Branch

The following is the syntax of branch flow control in RoB-MALLET:

BranchMalletProcess ::= "(" "IF" "(" "COND" Pred+ ")" MalletProcess MalletProcess? ")"

A *BranchMalletProcess* consists of two branches of *MalletProcess*, true and false. False branch could be empty. The evaluation of the condition is based on the mutual beliefs of involved agents. If the involved agents mutually believe that the condition is true, the *MalletProcess* in true branch is executed; otherwise false branch if there is.

For example, suppose that agent ag1 executes the following branch statement and agent ag1 fills role r1.

```
(IF (COND (case 1))
    (DO r1 (op1))
    (DO r1 (op2))
)
```

If condition `(case 1)` is true, ag1 executes `op1`; otherwise `op2`.

Our semantics of the condition in a branch flow control guarantees that all involved agents have a globally consistent decision on which branch to be executed, i.e., all agents only execute actions in a branch, either true or false branch, but not both. In current MALLET/CAST, conditions in branch flow controls are evaluated based on individual beliefs. Agents may have different individual beliefs on the condition; consequently, the involved agents may have inconsistent decisions on which branch to be executed, i.e., the agents may executes actions in both branches.

Conditions are also used in other flow controls, such as loop, FOREACH, FORALL, and CHOICE constructs. Thus, the evaluation of conditions is a common problem for all these flow controls. Rather than give a deep discussion for each flow controls, we will discuss the evaluation of conditions later in a separate section (Sec. III.5) and the semantics of conditions discussed in Sec. III.5 is applied to all flow controls that contain conditions.

III.4.4 Loop

The following is the syntax of loop flow control in RoB-MALLET:

LoopMalletProcess ::= "(" "WHILE" "(" "COND" Pred+ ")" MalletProcess ")"

A *LoopMalletProcess* consists of a *MalletProcess*. The evaluation of the condition is based on the mutual beliefs of involved agents. If the involved agents mutually believe that the condition is true, the *MalletProcess* is executed; otherwise the loop process is finished.

For example, suppose that agent ag1 executes the following loop statement and agent ag1 fills role r1. While condition `(case 1)` is true, ag1 repeats executing `op1`;

```
(WHILE (COND (case 1))
    (DO r1 op1)
)
```

Our semantics of the condition in a loop flow control guarantees that all involved agents have a globally consistent decision on whether the agents continue to execute the actions in the loop, i.e., all agents execute the actions, or all agents quit the loop. In current MALLET/CAST, conditions in loop flow controls are evaluated based on individual beliefs. Agents may have different individual beliefs on the condition; consequently, the involved agents may have inconsistent decisions on whether the agents continue to execute the actions in the loop. It is possible that some agents continue to execute the actions but other agents quit the loop. More detail about the evaluation of conditions will be given in Sec. III.5.

III.4.5 Other Flow Controls

Some additional flow controls are added into RoB-MALLET for easier use. Those flow controls include FOREACH, FORALL, and CHOICE. These constructs are not the focus of this dissertation, so we do not take these constructs in account in later chapters and they have not been implemented in our teamwork architecture RoB-CAST currently. Here, we just give brief descriptions about them. The reason why we bring these constructs is that we want to demonstrate that our RoB-MALLET is still under

development and additional constructs may be included later to accommodate new ideas. For more details, please refer to [28]. The following is the syntax of those flow controls:

ForEachMalletProcess ::= "(" "FOREACH" "(" "COND" Pred+ ")" MalletProcess ")"

ForAllMalletProcess ::= "(" "FORALL" "(" "COND" Pred+ ")" MalletProcess ")"

ChoiceMalletProcess ::= "(" "CHOICE" MalletProcess+ ")"

In a FOREACH construct, the condition may have multiple bindings. The involved agents execute the *MalletProcess* sequentially over these bindings. In a FORALL construct, the condition may have multiple bindings. The involved agents execute the MalletProcess concurrently over these bindings. More details about FOREACH and FORALL are available in [28]. As the conditions in branch and loop constructs, conditions in FOREACH and FORALL are evaluated based on mutual beliefs of all involved agents.

A CHOICE process allows agents to execute a list of *MalletProcess*es in order until one completes successfully. If the last process in the CHOICE process list fails, then the entire choice process fails. In current MALLET, if a *MalletProces* is an operator invocation, it fails when its precondition is not satisfied; if a *MalletProces* is a plan invocation, it fails when its precondition is not satisfied or its termination conditions is satisfied. Such definition of failure may not complete, but our RoB-MALLET keeps this interpretation given that our main concerns are not on specific constructs. More details about CHOICE are available in [28].

III.5 Conditions in Control Flows

Conditions are used by used in various constructs of flow controls in both RoB-MALLET and current MALLET, such as branch, loop, FOREACH, and FORALL constructs. We have given the syntax of these constructs in Sec. III.4.

In this section, we first discuss the semantics of conditions of these constructs in current MALLET/CAST and its problems; second, we analyze various issues related to the semantics of conditions in these constructs; third, we propose a set of syntax which

extend these constructs and allows users to flexibly express flow controls; and at last, we give our semantics of conditions with current syntax of these constructs.

While RoB-MALLET uses roles and role variables in flow controls, MALLET uses agent variables and agents in flow controls. However, as all of these are eventually filled by agents, and as sections 5.1 and 5.2 discuss both MALLET and RoB-MALLET, we just use agents directly in this section to make the descriptions easier to follow.

III.5.1 Semantics of Conditions in Current MALLET/CAST

In current MALLET/CAST, the conditions in flow controls are evaluated based on agents' individual beliefs. Every agent evaluates the conditions separately based on its individual beliefs. The agents may have different beliefs on the conditions. Some agents may believe that a condition is true, but other agents believe that the condition is false. Even though a condition is believed by some agents, these agents may have different bindings to make the condition true. Consequently, agents may not have consistent execution and further joint intention cannot be formed to ensure the agents' joint actions.

Suppose that team t1 consists of agents ag1 and ag2, that `top1` is a team operator, and that `plan1` is a team plan. Suppose the agents execute the following branch statement.

```
(IF (COND (pred ?x ?y))
    (DO t1 (top1 ?x))
    (DO t1 (plan1))
)
```

Because agents ag1 and ag2 separately evaluate the condition `(pred ?x ?y)` based on their individual beliefs, the agents may have different evaluations of the condition. Consequently the agents may enter different branches. Suppose that ag1 enters the true branch and that ag2 enters the false branch. As explained in III.3.5, an invocation of team operator/plan is a joint intention. As the agents enter different branches, the agents can form neither a joint intention to team operator `top1` (because

only ag1 invokes `top1`) nor a joint intention to team plan `plan1` (because only ag2 invokes `plan1`). Therefore, neither `top1` nor `plan1` is executed.

Even though both agents enter a branch, they still may not form a joint intention. Suppose that ag1 believes `(pred 1 2)` and that ag2 believes `(pred 3 4)`. Both ag1 and ag2 enter true branch to execute team operator `top1`. However, the team operators that they want to execute are different. Agent ag1 wants to execute `top1(1)` because the binding of `?x` for ag1 is 1 while Agent ag1 wants to execute `top1(3)` because the binding of `?x` for ag2 is 3. Therefore, agents ag1 and ag2 still cannot form a joint intention to execute `top1`.

Similarly, agents that execute loop, FOREACH, and FORALL constructs may not have consistent execution. For a loop construct, some agents may continue to execute a loop, but other agents may quit the loop. Even though the agents continue the loop, they may have different bindings for iteration. Further, agents cannot form joint intentions to execute joint actions in the loop. For a FOREACH or FORALL construct, given a binding, some agents may believe that the condition of the construct with the binding is true, but other agents may not believe that it is true. Further, agents cannot form joint intentions to execute joint actions in the construct.

III.5.2 Consideration on Semantics of Conditions

Based on the analysis in last section, the key to execute the teamwork specified control flows is that the involved agents must have a consistent execution with consistent bindings. To maintain consistent execution with consistent bindings, it is necessary for us to define a clear semantics about the evaluation of conditions in control flows. Moreover, the agents must execute the actions consistently based on the evaluation. The clear semantics about the evaluation of conditions must address which beliefs the evaluation is based on.

To clarify the semantics about which beliefs the evaluation of conditions in flow controls is based on, we define two terms, relevant individual and irrelevant individual. If an individual does not appear in any *MalletProcess* of a flow control, it is called an

irrelevant individual; otherwise, it is called a relevant individual. Also, an action in a *MalletProcess* of a flow control, is called relevant action to that flow control. Suppose there is a branch statement `(IF (COND (eq a b)) (DO ag1 (op1)) (DO ag2 (op2)))`. Agents ag1 and ag2 are relevant individuals but agent ag3 is an irrelevant individual. And, actions `(DO ag1 (op1))` and `(DO ag2 (op2))`are relevant actions.

With this distinction, we give all options about which belief the evaluation of the condition in a flow control is based on:

- The evaluation of the condition is based on the existence of a mutual belief on the condition among a set of relevant individuals. For example, a couple goes shopping if the wife is happy, i.e., the decision on whether the couple goes shopping is based on the evaluation of the wife's emotion. The set could contain either all relevant individuals or some relevant individuals. If the set only contains an individual, the evaluation actually turns to be based on the individual's beliefs.
- The evaluation of the condition is based on the existence of a mutual belief on the condition among a set of irrelevant individuals. If the set only contains an individual, the evaluation is actually based on the individual's beliefs. For example, a manager might evaluate the performance of a system and decide whether subordinate *SA* should work on algorithm improvement if the performance is poor, or subordinate *SB* should test his system.
- The evaluation of the condition is based on the existence of a mutual belief on the condition among a set of relevant and irrelevant individuals. For example, a manager (*MA*) together with a senior software engineer (*SN*) evaluates the performance of a system and decides that the SN should guide a software engineer (*SE*) to work on algorithm improvement if poor performance, or that the should *SN* guide a software tester (*ST*) to test the system otherwise.

III.5.3 Extensions of the Constructs of Flow Controls

With the current syntax of flow controls, we cannot distinguish which option listed in the above section is applied and how the set of individuals is determined. More information about on whose belief(s) the evaluation of the condition is based should be added. The flow controls may be modified to

BranchMalletProcess ::= "(" "IF" "(" "COND" Pred+ "BELIEVED" "BY" ByWhomSpec ")" MalletProcess MalletProcess? ")"

LoopMalletProcess ::= "(" "WHILE" "(" "COND" Pred+ "BELIEVED" "BY" ByWhomSpec ")" MalletProcess ")"

ForEachMalletProcess ::= "(" "FOREACH" "(" "COND" Pred+ "BELIEVED" "BY" ByWhomSpec ")" MalletProcess ")"

ForAllMalletProcess ::= "(" "FORALL" "(" "COND" Pred+ "BELIEVED" "BY" ByWhomSpec ")" MalletProcess ")"

ByWhomSpec ::= RoleName | RoleVariableName | "(" (RoleName | RoleVariableName)+ ")"

In the above constructs, the information about which beliefs the evaluation is based on is decided by the individuals listed after "BELIEVED BY". If a condition is believed by an individual, it can be interpreted as that its evaluation is based on the individual's belief. If a condition is believed by a set of individuals, it can be interpreted as that its evaluation is based on their mutual belief.

III.5.4 Semantics of Conditions in RoB-MALLET for Current Syntax

For the reasons of compatibility, our RoB-MALLET does not change the syntax condition in current MALLET[5]. To have a globally consistent execution, some assumptions must be made. In RoB-MALLET, we assumed the evaluation of a condition is based on the mutual beliefs of all relevant individuals and one of them is selected to be

[5] It is not hard to make such change. However, we will run experiments to compare our teamwork architecture, RoB-CAST, which uses RoB-MALLET, with previous teamwork architecture, CAST 3.0, which uses MALLET. We would like to run equivalent plans on these two teamwork architectures even though the plans are written in RoB-MALLET and MALLET.

a coordinator to evaluate the condition based on their mutual belief. This assumption basically follows option (1) in Sec. III.5.2 and the set of individuals is all the relevant individuals. Once the syntax of conditional constructs is modified as the above, we will have freedom to express all options with the clear semantics that leads to a globally consistent execution.

III.6 Summary

As a teamwork language, RoB-MALLET provides not only a set of constructs to express individual agent knowledge, including beliefs, goals, individual operators, individual intentions, and agents capabilities, but also a set of constructs to explicitly express team knowledge, including team structure, mutual beliefs, shared goals, team operators, JOINT DO's and team plans, and joint intentions. Moreover, RoB-MALLET can explicitly express complex actions with strong coordination by team operators. RoB-MALLET also extends team operators to JOINT DO's so as to express asymmetric complex actions with strong coordination.

The semantics of RoB-MALLET are consistent with teamwork theories. In RoB-MALLET, joint intentions can be explicitly express and the semantics of joint intention is consistent with Cohen and Levesque's joint intention. Also, RoB-MALLET reconciles Grosz and Kraus' shared plan theory and Jennings' commitment and convention. Joint intentions can be specified in terms of goals (similar to *Intend-that* in shared plan theory) or actions (similar to *Intend-to* in shared plan theory). Plans in RoB-MALLET can be either complete (similar to full shared plans) or incomplete (similar to partial shared plans). Similar to Jennings' conventions, termination conditions in team plans are used to monitor the execution of team plans.

Moreover, the semantics of RoB-MALLET are based on the mental states underlying teamwork. The relationships between the mental states have been clarified in the semantics of various RoB-MALLET constructs. In particular, RoB-MALLET relates the evaluation of preconditions of team operators, the evaluation of preconditions and termination conditions of team plans, the evaluation of conditions in various constructs of flow controls to team mental states, mutual beliefs, so that agents have a globally

consistent execution and further teamwork is enforced (i.e., joint intentions in control flows are ensured).

CHAPTER IV
ROLES AND ROLE-BASED PLANS

All the world's a stage,
And all the men and women merely players:
They have their exits and their entrances;
And one man in his time plays many parts,
His acts being seven ages. At first the infant,
Mewling and puking in the nurse's arms,
And then …
(W. Shakespeare, *As You Like it*, Act II, Scene 7)

IV.1 Introduction

From a perspective of human behaviors, Shakespeare metaphors that the social life is the stage of a theater on which men and women are acting roles. Similar to the social life, the teamwork is the stage of a theater on which agents are acting roles.

In the dictionary of [117], teamwork is defined as, 1) cooperative effort by the members of a group or team to achieve a common goal, 2) work done by a team, as distinguished from that done by personal labor, 3) work done by a number of associates, usually each doing a clearly defined portion, but all subordinating personal prominence to the efficiency of the whole; such as, the teamwork of a football eleven or a gun crew. These definitions of teamwork characterize the behavioral property of teamwork: teamwork is essentially cooperative "personal labors".

We introduce a set of concepts related to role to organize the cooperative "personal labors" of teamwork [13]. In the field of role, there are many interpretations of role. Based on the interpretations that capture the behavioral aspects, we define our concepts in a format amenable to computation. To characterize a collectively recognized category of persons who exhibit a set of behaviors, we define a position as a set of primitive operations. To represent the performance of behaviors, we define a role as an entity of a

position associated with a bag of temporally ordered actions. To capture dynamic association with roles, we define a role variable as an entity that is dynamically selected from a set of entities to be associated with a bag of temporally ordered actions. In our teamwork architecture, the behavioral knowledge of teamwork is represented by team plans, and we specify team plans (called role-based plans) in terms of roles and role variable.

In addition to expressing the behavioral knowledge of teamwork, our concepts of role and role variable enable our mechanisms of task decomposition and task delegation, by which the behavioral knowledge of teamwork specified by role-based plans drives agents to actually execute teamwork.

Our mechanism of task decomposition is based on a notion of responsibility, which is defined in terms of what a responsibility contains and how a responsibility impacts the mental states of the agent(s) taking the responsibility. The team responsibility of all the roles in a role-based plan contains all temporally ordered actions associated with the roles and their impact on the mental states of the agents (*T*) taking the team responsibility. The individual responsibility of a role (or role variable) contains the temporally ordered actions associated with the role (or role variable) and their impact on the mental states of the agent taking the responsibility graph. A team task is translated to a team responsibility. Through decomposing the team responsibility graph to individual responsibility graphs, a team task is decomposed to individual sub-tasks.

Our mechanism of task delegation is realized by delegating individual responsibility to agents. In order to execute a role-based plan *P*, an admissible team assignment is required to delegate all roles in a role-based plan to the agents invoking the role-based plan so that the agents intend to execute the actions in the individual responsibilities of the roles. During the execution of the role-based plan *P*, once a role is selected to fill a role variable, the agent to which the role delegated intends to execute the actions in the responsibility of the role variable.

In Sec. IV.2, we will define our concepts of position, role and role variable. In Sec. IV.3, we will define role-based plan in terms of roles and role variable and give the

RoB-MALLET syntax related to position, role, role variable and role-based plan. Also, we will explain how actions can be express in the processes of role-based plan in terms of roles and role variables. In Sec. IV.4, we will define the notion of responsibility, including the responsibility of a role, the responsibility of a role variable and the team responsibility of a virtual team of roles. We will introduce a graphic language to represent responsibilities. Moreover, we will present algorithms for translating a role-based plan into a team responsibility graph. In Sec. IV.5, we will describe our method for decomposing a team responsibility graph to individual responsibility graphs. The algorithm corresponding to the method will be also presented. In Sec. IV.6, we will describe our mechanism of task delegation. The individual responsibilities of roles are delegated to agents through team assignments. The individual responsibilities of role variables are delegated agents through role selections. The admissibility of team assignment will be formalized as a constraint satisfaction problem (CSP). We will present a CSP algorithm to search for admissible team assignments. In Sec. IV.7, we will describe how role-based plans can be used as actions in planning algorithms, including forward state-space search and backward state-space search. Also, we will present an algorithm to search a role-based plan for achieving a goal. With slight modification, the algorithm can be used in planning algorithms to check which role-based plans are applicable to a state. In Sec. IV.8, we will discuss the difference between agent variable used in MALLET/CAST and our roles and role variables, and give a summary to this chapter.

IV.2 Concepts Related to Role

In this section, we will define a set of concepts including position, role and role value to capture the behavioral aspects of the general term of role. We ground our definitions on Biddle and Thomas' classificatory concepts of role, some of which capture the behavioral aspects of role, in their role theory [4]. We sort out those aspects that are not amenable to our computational purpose and come up with three concepts, including position, role and role variable, to characterize behavioral aspects of the general term of

role. We will give this rationale in Sec. IV.2.1. Then, we will formally define position, role and role variable in Sec. IV.2.2, Sec. IV.2.3 and Sec. IV.2.4 respectively.

While behaviors in Biddle and Thomas' classificatory concepts of role could be complex actions, we take Grosz and Kraus' view that complex actions must be rested eventually in the primitive operations of individual agents [36, 37]. Therefore, our formalisms are based on primitive operations (i.e., individual operators in RoB-MALLET). In Sec. IV.2.5, we will clarify how complex actions can be broken into primitive operations so that our formalisms accommodate complex actions.

IV.2.1 Rationale

The concept of role is generally but loosely understood as [117]: a character assigned or assumed, a socially expected behavior pattern usually determined by an individual's status in a particular society, a part played by an actor or singer, a function or part performed especially in a particular operation, or an identifier attached to an index term to show functional relationships between terms. This understanding implies an attractive property of roles: a role stands for a functional abstraction, independent of the specific person who plays it.

To simulate teamwork, we need some concepts to help us organize team activities, particularly the concepts should allow us to specify team activities conceptually so that the knowledge about the team activities can be used by different agents; we also need some mechanisms to decompose team activities into individual activities and delegate the individual activities to agents so that team activities can be actually executed. The property of roles that described above makes roles suitable to help us to realize these goals. In order to use concepts related to role for these purposes, we need to define the concepts to capture the behavioral aspects of the general term of role.

We ground our concepts related to role on a well-founded theory, Biddle and Thomas' role theory [4]. Biddle and Thomas investigated a variety of speculations, hypotheses and theories about particular aspects of role from different perspectives, such as philosophy, anthropology and sociology. In [4], they gave classificatory concepts of role based on a set of phenomenal referents, including behaviors, persons, or persons and

their behaviors. The behavioral aspects of roles are captured by the referent of behaviors and the referent of persons and their behaviors, while the referent of persons is not related to the behaviors aspects of role. Our interest is to define a formalism of role amenable to computation, particularly specifying team activities at abstract level independent of the specific agents that will execute them, decomposing team activities into individual tasks, and delegating the individual tasks to actual agents. So we will sort out those non-behavioral concepts and derive our concepts from their classificatory concepts amenable to our purpose.

Biddle and Thomas concluded that there are several distinct bases upon which a notion of roles can be defined.

- Roles can be defined based on partitioning concepts for behaviors. A role defined by this referent characterizes the exhibit of behaviors (i.e., the ability of behaviors). For example, a role of secretary can be defined as secretary's behaviors: answering phones, receiving faxes, filing documents, and arranging schedules. Although this definition associated roles with behaviors, it does not characterize the performance of the behaviors, i.e., when and why behaviors are performed.
- Roles also can be defined based on partitioning concepts for persons, for example, father and son.
- Roles also can be defined based on partitioning concepts for persons and their behaviors. The distinction between this referent and the first one is, the behaviors in this referent focus on the dynamic aspects (i.e., the performance of behaviors), and those in the first referent focus on the static aspects (i.e., ability of behaviors). Given a role defined by this referent, its behaviors are combined with a specific person filling the role and following a specific procedure, which leads to concrete performance of (some or all of) the behaviors. Generally speaking, a role defined by this referent is an entity of a position, which executes specific behaviors of the position. For example, a role of a position secretary for a CEO is

the one who is associated with specific actions, such as arranging the CEO's schedules by following some specific routine.

The referent of partitioning concept for persons is a non-behavioral criteria and thus not suitable for utilization in a formal role concept suitable for computation. Even though the referent of partitioning concepts for behaviors, alone, does not directly lead to concrete actions, it can characterize the behavior requirements of roles (i.e., interpreted as "job qualifications"). The referent of partitioning concepts for persons and their behaviors is suitable to our purpose: specifying team activities conceptually and helping decompose and delegate team activities to agents. Therefore, based the referent of partitioning concepts for behaviors, we define a position as all entities who can perform a set of primitive operations, and use this to characterize a collectively recognized category of persons who are able to exhibit a set of behaviors. Then, based on the referent of partitioning concepts for persons and their behaviors, we define a role as an entity of a position associated with a bag of temporally ordered actions. A role, as an entity of a position, must be capable of the primitive operations of the position. We note that a role is more than an entity of a position, in that it includes an (partial) ordering on a (sub)bag of actions in the position.

Biddle and Thomas [4] also pointed out that pre-association with roles is too restrictive. There is yet another level of abstraction which we wish to introduce that mirrors a concern that has arisen in the role theory literature. They suggested that some actions are not predetermined to be firmly associated with specific roles. Rather, they argue that a specific action might be performed by any one of several roles, with the determination of which role actually performs it being determined dynamically in a specific situation.

Our analogy to this is to introduce role variables. We define a role variable as an entity which is dynamically selected from a set of roles to be associated with a bag of temporally ordered actions. Generally speaking, a role variable stands for some role out of a set of roles. Depending on concrete situation, one role from the set is dynamically selected to fill (or assume) the role variable. When this happens, the bag of operations

of the assuming role is dynamically expanded. For example, in plan `killwumpus(?w)` (discussed in Sec. III.3.4.3), a role variable ?fi is defined to kill the found wumpus ?w. To kill the wumpus ?w faster, one which of role r2 and r3 is closer to the wumpus ?w is dynamically selected to be ?fi.

IV.2.2 Position

Based on the referent of partitioning concepts for behaviors, we define a concept of position to refer a collectively recognized category of entities (persons) that exhibit a set of behaviors. An entity is an abstraction of any performer (e.g., agent or person) that is able to perform operations.

Formally, a position is defined as a named set of all entities (e.g., agents or persons) that are able to perform a set of primitive operations, denoted by operators. Given a set of operations *O*, a position based on the operations is

$$RC(O) = \{e \mid e \in \textbf{Entities} \ \wedge \forall op\ (op \in O) \wedge \textbf{\textit{Capable}}(e, op)\ \},$$

where *O* is a set of operators, *e* represents an entity capable of performing actions, and ***Capable***(*e*, *op*) means that *e* is able to perform *op,* assuming the preconditions of *op* are true. The set of operations *O* is also called the qualification of the position.

The purpose of defining position is to capture the capability requirements on agents. In later section, a role is defined as an entity of a position. To fill the role, an agent is required to be able to perform all operations defined by the position. More details will be given in later sections/chapters.

Note that we have not required that positions be disjoint. Thus, there can be entities capable of performing the operations associated with multiple positions. This fact will be used later (Sec. IV.6) in being able to delegate roles to agents that belong to different positions.

The RoB-MALLET syntax of position will be given in Sec. IV.3.2.

IV.2.3 Role

Based on the referent of partitioning concepts for persons and their behavior, we define a role to be an abstraction of an entity (agent) that performs a specific bag of the

actions[6] and includes temporal constraints on the order in which the operations may be performed.

An action may be either unconditional or conditional. If it is unconditional, it must be performed; otherwise, it is performed only when the condition φ is true. We assume that that every entity of a position can evaluate the condition φ.

We base our definition for a role on a position. In other words, any action (i.e., primitive operation) associated with the role must be in the operation set of the position. Later it will be a requirement that any agent taking that role must be able to perform all of the operations of the position (whether or not those operations are actually called for in the specific operations of the role).

Now, we give a definition of a role. Let O be a bag of operations that must be performed by the same entity of a position, *COND* be a bag of conditional operators where each action is contingent on a conjunction of conditions, and C_O be a bag of ordering constraints that impose a "temporal order" on O and the evaluation of the conditions in *COND*. Let *RC* be a position and O_{RC} the qualification of *RC*. We define a role r as an abstraction of the entity of position *RC* that satisfies the constraints. We denote a role as a tuple:

$r = (id, RC, O, COND, C_O)$,

where *id* is the name of r and refers to the entity of RC that must perform the operations, $O \subseteq O_{RC}$, C_O is a set of temporal orders as (o_i, o_j) where $o_i, o_j \in O \cup COND \cup S$, and $S = \{o_s, o_e\}$. o_s is a dummy starting operation that can be performed by any entity of a position, and o_e is a dummy ending operations that can be performed by any entity of a position. (o_i, o_j) means that o_j occurs after o_i and o_j can occur only after o_i has occurred. Moreover, if o_i is an operator dependent on a condition φ, there are two special ordering constraints, (φ, o_1) and $(\varphi, o_2)_\neg$. (φ, o_1) means that o_1 is performed if φ is true. $(\varphi, o_2)_\neg$ means that o_2 is performed if φ is false. A condition φ could be constrained by both of

[6] We note that, for simplifying our formalization, actions in our formalisms of role and role variable are primitive operations (i.e., operators in RoB-MALLET). However, in Sec. IV.2.5, we will clarify that complex actions can be rested in primitive operations and how the formalisms accommodate complex actions. After Sec. IV.2.5, when we mention the actions associated with roles or role variables, the actions could be either primitive operations or complex actions.

these two special orderings. If so, they represent true and false branches of the condition φ.

Temporal orders are transitive. That is, if there are two temporal orders (o_i, o_j) and (o_j, o_k), then (o_i, o_k) is a true temporal order too. However, we note that C_O is not a transitive closure. That means, even though (o_i, o_j) and (o_j, o_k) are in C_O, (o_i, o_k) may not be in C_O.

And, cycles may exist in the set of temporal orders C_O. That is, there exist temporal orders as (o_1, o_2), (o_2, o_3), …, (o_k, o_1), where $k \geq 2$. This does not mean a temporal conflict. Rather, it means that the operators may be executed more than once in accordance with the temporal orders. The feature also holds when we talk about the temporal orders in role variables (described later in Sec. IV.2.4) and in the process of a role-based plan (described later in Sec. IV.3.1.1).

Operationally, we will declare role names to be associated with positions and then define the processes of team plans (see Sec. IV.3.3) in terms of actions performed by the role (names). The operations associated with a role will be derived as a plan is executed. Thus, the specific operations and constraints in O and C_O are dynamically determined as the execution of a system progresses. Roles are thus dynamically determined. However, the static nature of the position underlying a role allows constraints to be imposed that assure that any agent taking on a role will be capable of performing the operations of the role.

In addition to the temporal order applied to the operations of role r, there can be temporal orders between the operations of role r and those of other roles (or role variables, discussed in Sec. IV.2.4). For example, it might be required that operation o associated with role r appear in order before operation o' in role r'. The above definition thus needs to be extended to deal with this situation. Let O_X be the union of the sets of operations (in other roles) involved in temporal ordering constraints with operations of the role r (including evaluation of conditions and dummy starting operations). Let C_X be the temporal order formed to express the ordering constraints between the operations in

the union of O, $COND$ and S and those in O_X, that is, C_X is a set of temporal orders as (o_i, o_j), where $o_i \in O \cup COND \cup S$ and $o_j \in O_X$, or $o_j \in O \cup COND \cup S$ and $o_i \in O_X$.

The ordering constraints of role r are then extended to include C_X. We use $<_r$ to denote the extended ordering constraints as $<_r = C_O \cup C_X$. Therefore, the definition of a role r is extended to $r = (id, RC, O, COND, <_r)$.

Later (see Sec. IV.6) we will develop mechanisms for real agents to take on one or more roles (which we will think of in terms of responsibility). An important characteristic of our notion of role is that it includes the specific operations that an agent filling the role must perform in a specific setting as well as their ordering constraints. Taking this view on role definition will enable us to reason about the assignment of real agents to roles, to talk about the responsibility of roles, to express automated helping behaviors and to achieve a certain level of plan reusability.

The RoB-MALLET syntax of role will be given in Sec. IV.3.2 and an example of a role-based plan will be given in the end of Sec. IV.3.2 to show how roles are declared and how roles are associated with temporal actions.

IV.2.4 Role Variable

We define a role variable as an abstraction of an entity that is dynamically selected from a set of roles to be associated with a bag of temporally ordered actions. An example of a role variable is ?fi in plan `killwumpus(?w)` (discussed in Sec. III.3.4.3). A role variable is similar to a role except that the role variable is dynamically selected from a set of roles. So, in the definition of a role variable, we do not explicitly require it to belong to any specific position. Rather, we require a selection constraint whose satisfaction will select one role out of a set of roles[7] to perform the operations of the role variable in accordance with the specified order. For clarity, we will prefix role variables by "*?*". Then we define a role variable *?r* as a tuple:

[7] One may question the possibility to include role variables in the selection scope. A role variable is to select a role so as to associate the actions associated with the role variable to the role. If role variables are included in the selection scope and a role variable *?rv1* is selection, essentially the role that is selected to fill *?rv1* is selected. From the perspective of associating actions to role, it is equivalent to just including roles in the selection scope. Thus, we only allow roles in the selection scope.

$?r = (?id, O, COND, C_O, RS, SC)$,

where ?*id* is the name of *?r*, *RS* is the selection scope, i.e., a set of roles, *SC* is the selection constraint, and for any $r \in RS$ and if $r \in RC$, $O_{RC} \supseteq O$.

In the definition of role variable, the first four elements have the same meanings as those (with the same symbols) in the definition of role. They define the actions associated with the role variable, the conditions on the actions, and the temporal orderings on the actions. Moreover, the definition of role variable includes the selection scope *RS* and the selection constraints *SC*. The selection scope *RS* defines all possible roles that can be selected to be the role variable. The selection constraint SC defines the conditions that the role must satisfy for being selected to the role variable. To select a role to fill the role variable, all roles in the selection scope (eventually agents) coordinate with each other, and the one that satisfies the selection constraint is selected. If more than one role satisfies the selection constraint, they coordinate with each other and make sure one of them is selected (we assume such selection is random). If no role satisfies the selection constraint, then the action associated with the role variable cannot be associated with any role and thus they cannot be executed. This may cause a failure in the context of a set of roles (discussed later in role-based plans). We assume that users need to ensure that at least one role will satisfy the selection constraint when specifying the role variable, or capture the failure caused by no role satisfying the selection constraint by termination conditon. We note that conditions in *COND* have exactly the same semantics as those in the definition of role for defining conditional actions instead for constraining role selections. Through the role variable, the actions associated with the role variable are dynamically associated with the role that is selected to fill the role variable.

Similar to roles, there can be temporal orders between the operations (including evaluation of conditions and dummy starting operations) of the role variable and those of other roles and role variables. Let O_E be the union of the operations of the other involved roles and role variables. A temporal order C_E is formed to express the ordering constraints between the operations in the union of *O*, *COND* and *S* and those in O_E, that

is, C_E is a set of temporal orders as (o_i, o_j), where $o_i \in O \cup COND \cup S$ and $o_j \in O_E$, or $o_j \in O \cup COND \cup S$ and $o_i \in O_E$.

It follows that the ordering constraints of role variable *?r* are extended to include C_E. We use $<_{?r}$ to denote the extended ordering constraints as $<_{?r} = C_O \cup C_E$. Therefore, role variable *?r* is defined as *?r* = (*?id*, *O*, *COND*, $<_{?r}$, *RS*, *SC*).

The RoB-MALLET syntax of role variable will be given in Sec. IV.3.2 and an example of a role-based plan will be given in the end of Sec. IV.3.2 to show how role variable are declared, how role selections are specified, and how role variables are associated with temporal actions.

IV.2.5 Accommodating Complex Actions

In the above formalisms, actions associated with roles and role variables are primitive operations. However, behaviors in Biddle and Thomas' classificatory concepts of role can be either primitive operations or complex actions. Correspondingly, actions associated with roles and role variables can be complex actions too. Rather than re-formalize our roles and role variables to directly include complex actions, we break complex actions into sub-actions, eventually into primitive operations, and associate the primitive operations to individuals (roles or role variables).

While other researchers [109, 114] argue complex actions may include many different aspects and could be specified in other ways, a complex action in which an individual is involved together with other individuals could appear as the following in our teamwork architecture:

- Coordinated operations: the operations of a role (or a role variable) are coordinated with the operations of other roles (or role variables). As described in Chapter III, such coordinated operations could be either team operators or joint do, and coordination modes include AND, OR, and XOR;

- Plan invocations: a role[8] forms a sub-team (with other roles, perhaps just the role itself) to invoke a special type of complex action, a plan (the syntax of plan has been described in Chapter III and more details about role-based plan will be given in Sec. IV.3), which consists of a process of actions.
- Role selections for role variables: the action(s) associated with a role variable might be performed by any one of a set of roles. The roles need to negotiate to decide which one is selected to fill the role variable and thus perform the actions.

Other type of complex actions may be developed; however, we will only focus on the above types of complex actions in later sections of the dissertation, including how to represent them and how to break them into primitive operations.

Suppose $r_i = (id_i, O_{ri}, Cond_{ri}, <_{ri})$, $?r_k = (?id_k, O_{?rk}, Cond_{?rk}, <_{?rk}, RS_{?rk}, SC_{?rk})$, and $r_j = (O_{rj}, Cond_{rj}, <_{rj})$. Complex actions, including coordinated operations, plan invocations and role selections, are broken eventually into primitive operations and the primitive operations dynamically associated with the involved roles (or role variables) as follows:

- role r_i (or role variable $?r_k$) involves a coordinated operation *co*. In RoB-MALLET, coordinated operations could be team operators or joint do (described in Chapter III). We assume there is an individual operator corresponding to each team operator and such an individual operator is associated with each involved role. For example, the individual operator "lift-table" corresponds to team operator "lift-table" and an individual operator "lift-table" is associated with r_i. For a joint do, the individual operators and their association with the involved roles are specified in the joint do statement. Role r_i (or role variable $?r_k$) needs to dynamically coordinate with each other according to its coordination mode to decide how to perform the individual operation *io*. Based on the result of

[8] In our teamwork architecture, we just disallow role variables to invoke team plans. One may question that this may pose limitation on the invocation of sub-plan. A role variable has a set of constraints to select a role and eventually an agent. A role-based plan also contains a set of constraints on delegating roles to agents. If a RoB-MALLET user wants the role variable to invoke the role-based plan, the user can just walk around by merging the constraints for role selection with the constraints for delegating roles to agents and just let the roles in the selection scope invoke the plan.

coordination, if r_i needs to perform *co*, then r_i includes *io* in its *O* and replaces *co* with *io* in its $<_{ri}$;

- role r_i involves plan invocation *a*, which consists of a process of actions in terms of a set of roles, *VT'*. The plan invocation *a* is broken into the actions and eventually into primitive operations. Without losing generality, we can assume the actions are primitive operations. However, it is not decided which part of the actions is associated with r_i (in other words, which role(s) of *VT'* are filled by r_i) until involved roles negotiate. Suppose r_i is selected to fill a role r_j in *VT'*. The actions associated with r_j are dynamically included into r_i as

  $r_i \rightarrow r_j \Rightarrow r_i = (id_i, O_{ri} \cup O_{rj}, Cond_{ri} \cup Cond_{rj}, <_{ri} \cup <_{rj})$,

  where $\rightarrow$ means "is selected to fill". Moreover, after r_i has completed the actions, they should be dynamically removed from r_i. Because r_i and r_j are at two different levels of the plan hierarchy, the bags O_{ri} and O_{rj}, $Cond_{ri}$ and $Cond_{rj}$, and $<_{ri}$ and $<_{rj}$ do not overlap. Even though they are in the same plan definition, they are in two different plan invocations. So they there is no conflict in their unions. In this way, a plan invocation as a complex action can be eventually decomposed into individual operations and dynamically associated with the involved roles;
- role r_i involves a negotiation regarding action association of role variable $?r_k$. That is, it is not decided to which role the actions of $?r_k$ are associated until involved roles negotiate. Suppose r_i is selected to fill $?r_k$, the actions associated with $?r_k$ are dynamically included into r_i as

  $r_i \rightarrow ?r_k \Rightarrow r_i = (id_i, O_{ri} \cup O_{?rk}, Cond_{ri} \cup Cond_{?rk}, <_{ri} \cup <_{?rk})$,

  where $\rightarrow$ means "is selected to fill". Moreover, after r_i has completed the actions, they should be dynamically removed from r_i. In this way, any negotiation on action association as a complex action can be expressed by individual operations.

We note that the primitive operations, which the complex actions eventually rest on and are associated with a role, are in the operation set of the position of the role. Also, to break complex actions to sub-actions and eventually to primitive operations, the involved individuals need to negotiate with each other. We assume that all entities can

communicate with others by sending and receiving messages so that negotiations can take place.

IV.3 Role-based Plans

In this section, we present an abstract notion of a role-based plan and its components (in Sec. IV.3.1). In contrast to plans in MALLET, role-based plans are specified in terms of roles and role variables. To express role-based plans in RoB-MALLET, we will give the RoB-MALLET syntax and semantics related to role-based plans, including position, role declaration, role variable declaration, virtual team, constraints on delegating roles to agents, and plan (in Sec. IV.3.2). Also, we will explain how actions are associated with roles and role variables in the processes of role-based plans (in Sec. IV.3.3).

IV.3.1 Definition of Role-Based Plans

Similar to team plans described in Chapter III, a role-based plan also consists of a set of preconditions Φ, a set of effects Θ, a set of termination conditions Ψ, and a process of how to achieve the effects.

However, for ensuring team plans are reusable for different agents, the processes of plans need to be specified as conceptual descriptions of to achieve the effects in any situation in which the preconditions are satisfied, independent of the agents executing the plan. It is of crucial importance to make sure that real agents are not allocated to actions directly, but only through some mediating entities. The process of a role-based plan is specified in terms of roles or role variables, i.e., actions in the process are associated with roles and role-variables.

Corresponding to the adoption of roles and role variables in the process of a role-based plan, a role-based plan also contains a set of the names of all roles in the process (called virtual team), a set of the names of all role variables in the process, and a set of constraints on delegating roles to agents. As discussed in Chapter II, roles have been used in other teamwork architectures, such as STEAM. In those teamwork architectures, selecting agents for roles is based on capability requirements. In our teamwork architecture, in addition to capability requirements, constraints on delegating roles to

agents can be specified to express the relationships among delegating differents roles to agents. For example, two roles of the position scout might work in different zones. These two roles thus impose a condition on selecting agents: the agents assigned to the roles must be different for geographical reasons.

IV.3.1.1 Process in Role-Based Plans

In a role-based plan, the process is still the place to specify team activities. Role-based plans adopt the concepts of role and role variable to organize team activity. Role-based plans are composed by roles and role variables, i.e., actions in the process are associated with role and role variables. The details about how to specify actions in the process in terms of roles and role variables will be given in Sec. IV.3.3.

Suppose the process of a role-based plan is composed by a set of roles $r_1, r_2, \ldots, r_m$, where $1 \le i \le m$, $r_i = (RC_{ri}, O_{ri}, Cond_{ri}, <_{ri})$, and a set of role variables $?r_1, ?r_2, \ldots, ?r_n$, where $1 \le j \le n$, $?r_j = (O_{?rj}, Cond_{?rj}, <_{?rj}, RS_{?rj}, SC_{?rj})$. The process, denoted *Proc*, is the union of the roles and role variables.

$$Proc = (\bigcup_{i=1}^{m} O_{ri} \cup \bigcup_{j=1}^{n} O_{?rj}, \bigcup_{i=1}^{m} Cond_{ri} \cup \bigcup_{j=1}^{n} Cond_{?rj}, \bigcup_{i=1}^{m} <_{ri} \cup \bigcup_{j=1}^{n} <_{?rj}, \bigcup_{i=1}^{m} RS_{?rj}, \bigcup_{i=1}^{m} SC_{?rj})$$

As we described in Sec. IV.2.3 and IV.2.4, the temporal orderings on a role (or role variable) are extended to be between the actions of the role (or role variable) and the actions of other roles and role variables. For the roles and role variables in the process, such extension is limited within the actions in the process, i.e., the temporal orderings of a role (or role variable) are between the actions associated with the role (or role variable) or between the actions associated with the role (or role variable) and the actions associated with other roles and role variables in the process. As described in Sec. IV.2.5, the actions associated with roles in the sub-plan are dynamically associated with roles invoking the plan. So, this limitation still holds even though a sub-plan is invoked in the process.

IV.3.1.2 Roles and Role Variables in the Process

The difference between the actions associated with roles and actions associated with role variables is that, action associations by roles are static (or direct) while action associations by role variables are dynamic (or indirect). As described in Sec. IV.2.4, some actions may be performed by one of a set of roles and which role to perform the actions is determined dynamically by specific situations. In a role-based plan, such actions are associated with role variables and the potential roles are selected to fill the role variables. In this way, the process of a role-based plan allows both static and dynamic action association. Moreover, as explained in Sec. IV.2.5, by dynamic action association, the actions associated with role variables are dynamically associated with the roles which fill the role variables. Thus, even though there are both static and dynamic action associations, a role-based plan is still can be viewed as being composed by roles; and role variables can be viewed as an intermediate to express dynamic action associations. More details about action association will be given in Sec. IV.3.3.

We call the aggregation of the roles in a role-based plan *P* the *virtual team* of the plan, denoted by *VT*, *VT*(*P*) or *roles*(*P*). We call the aggregation of the the role variables in a role-based plan as role variable set of the plan, denoted by *RVS*, *RVS(P)* or *rolevars(P)*. We note that, roles in *VT* may form a sub-team to invoke a sub-plan and the sub-plan contains a virtual team too; however, the virtual team of the sub-plan is not part of *VT*. As explained in Sec. IV.2.5, the sub-plan invocation is a complex action and the actions associated with roles in the sub-plan are dynamically associated with the roles in *VT* which invoke the sub-plan.

IV.3.1.3 Constraints on Delegating Roles to Agents

Roles and role variables in role-based plans are only abstractions of entities to perform actions and specify team activities. They do not lead agents to perform team activities until they are assigned to the agents (i.e., the roles are delegated to the agents). As we explained earlier, role variables are only intermediates for dynamically associating actions to roles during the execution of role-based plans. In order to execute

team activities specified in a role-based plan, the roles in a role-based plan need to be delegated to the agents that invoke the role-based plan.

To delegate roles to agents, the agents must be "qualified" to fill the roles. A necessary condition for an agent being qualified to fill a role is that the agent must be able to perform operations required by the role. As well as the qualification of the roles, there can be a set of social norms (i.e., conditions), which specify the selection of agents for roles and the admissible relationships between roles. The social norms may exist for different kinds of reasons, such as geographical reasons, workload balance, cooperation requirement. For example, to kill a big wumpus, two roles of the position fighter need to perform team operator shoot. These two fighters must be filled by different agents.

In a role-based plan, we define social norms as a set of constraints on delegating roles to agents, and denote these constraints by Γ. Regardless the underlying reasons, the constraints are specified as a set of predicates. The users can identify the reasons and express them as constraints in proper way. For the reuse of role-based plans, the constraints are also specified in term of roles, instead of concrete agents. If a role r_i is involved in a constraint Γ_j, the constraint Γ_j is said to be *relevant* to r_i.

To evaluate the constraints, the roles used in the constraints need to be substituted by the agents that are going to be assigned to the roles. The agents selected to invoke a role-based plan must satisfy the constraints of the plan before they are can be assigned to roles and perform actions associated with the roles.

IV.3.1.4 Definition of Role-Based Plans

Finally, then, we formally define a role-based plan P as $P = (\Phi, \Theta, \Psi, VT, \Gamma, RVS, Proc)$, where Φ is a set of preconditions, Θ is a set of effects, Ψ is a set of termination conditions, VT is a virtual team[9], Γ is a set of constraints specifying social norms on delegating roles to agents, RVS is a set of role variables to specify non-predetermined actions, and $Proc$ is the process of actions of VT achieving the pursuing goal (i.e., effects) under similar situations (i.e., preconditions).

---

[9] Note that concrete agents may not be included in the plan definition.

If *VT* just contains a single role, *P* is an individual plan; otherwise, *P* is team plan. In the process of plan *P*, an action may be a plan invocation. Suppose that a role-based plan *P′* is invoked by a subset of roles of *VT*. We call *P* a hierarchical plan and *P′* sub-plan of *P*. A plan without sub-plan invocation is called flat plan.

IV.3.2 Related Syntax and Semantics in RoB-MALLET

In RoB-MALLET, the processes of plans are defined in terms of roles and role variables. We note that agents are explicitly excluded from the processes. In this section, we will describe the syntax and semantics of various constructs for specifying role-based plans, including position, role declaration, role variable declaration, virtual team, constraints on delegating roles to agents, and role-based plan.

IV.3.2.1 Position

In RoB-MALLET, a position is defined based on operators, either individual or team operators. The following is the syntax of position:

PostionDef ::= "(" "POSITION" PositionName "(" OperName ")" ")"*

As described in Sec. IV.2.2, every entity of a position is required to be capable to do the operators defined in the position. We consider the entities of the position to be the set of all agents able to perform all of the operations listed for the position. For example,

```
(POSITION sniffer (talk move movein randmove sense selectTarget))
```

A position sniffer is defined by a set of operators, including `talk`, `move`, `movein`, `randmove`, `sense`, and `selectTarget`. Suppose r1 is an entity that takes on the role sniffer; r1 should be able to perform the above operators.

IV.3.2.2 Role Declaration

As described in Sec. IV.2.3, a role is an abstraction associated with a sequence of actions and a role-based plan composes a set of roles. In RoB-MALLET, roles are defined in a plan and together with other roles to compose team activities of the plan.

The definition of a role includes two parts: role declaration and the association of actions with role.

In RoB-MALLET, a role is declared as an entity of a position with an identity. The following is the syntax of role declaration:

RoleDef ::= "(" "ROLE" "(" RoleName+ ")" PositionName ")"

For example, *(ROLE (r1) sniffer)* declares *r1* as an entity of position *sniffer*. *r1* is required to do operators talk, move, movein, randmove, sense, and selectTarget.

Even though a role is declared as an entity of a position, the declaration just characterizes the behaviors that entities filling the position must be able to perform, i.e., what operators the role may do. The actions associated with roles are specified in the process of the plan. We will describe how to associate actions with roles in Sec. IV.3.3.

IV.3.2.3 Role Variable Declaration

Similar to roles, the definition of a role variable contains two parts: role variable declaration (also called role selection) and actions association. Both role variable declarations and actions association are specified in the process. The declaration of a role variable must be specified before actions are associated with it. RoB-MALLET declares a role variable by the name of role variable, a subset of roles in the role-based plan, and a set of constraints as the criteria of selecting roles. The following is the syntax of role variable declaration in RoB-MALLET

SelectionMalletProcess ::= "(" "SELECT" RoleVariableName RoleList ConstraintsList ")"

RoleList ::= "(" RoleName+ ")"

ConstraintsList ::= "(" "CONSTRAINTS" Pred+ ")"

The roles which could be selected to fill a role variable are listed in *RoleList*. To be selected to fill a role variable, a role must satisfy the constraints in the declaration of the role variable.

Role variable declaration itself implies also a kind of cooperation among involved roles. It requires negotiation among the involved roles to decide which role is selected to perform the actions associated with the role variable. Therefore, role variable declaration

appears as an action in the process of a role-based plan and before the action associations with the role variable.

Similar to roles, the actions associated with role variables are specified in the process of the plan. We will describe how to associate actions with role variables in Sec. IV.3.3. Before actions associated with a role variable, the role variable must be declared.

IV.3.2.4 Virtual Team

The virtual team of a role-based plan is defined as the set of all roles in the plan. The following is the syntax of virtual team in RoB-MALLET:

RoleTeamDef ::= "(" "ROLETEAM" RoleTeamName? "(" RoleDef+ ")" ")"

IV.3.2.5 Constraints on Delegating Roles to Agents

The constraints on delegating roles to agents are defined by a set of predicates. The following is the syntax of constraints on delegating roles to agents in RoB-MALLET:

ConstraintsList ::= "(" "CONSTRAINTS" Pred+ ")"

The names of roles in virtual team of a plan may appear in the predicates. When delegating the roles to a team of agents, the predicates are evaluated after replacing the name of the roles with those of agents who want to fill the roles.

IV.3.2.6 Plan

A role-based plan is defined by a set of preconditions, a set of effects, a set of termination conditions, a virtual team of roles, a set of constraints on delegating roles to agents, and process. The following is the syntax of plan in RoB-MALLET:

PlanDef ::= "(" "PLAN" PlanName VariableListOpt RoleTeamDef ConstraintsList? PreConditionList? EffectsList? TermConditionsList? "(" "PROCESS" MalletProcess ")" ")"

Similar to constraints on delegating roles to agents, the names of roles in virtual team may appear in the predicates of preconditions, effects and termination conditions. To evaluate these predicates, the names of the roles should be replaced with those of agents who fill the roles.

The following code is an example that shows the syntax we described. It defines two positions (`sniffer` and `fighter`) and a role-based plan `scanandkill`. In the role-based plan `scanandkill`, the virtual team consists of three roles, r1, r2 and r3. r1 is a sniffer, and r2 and r3 are fighters. Also, a role variable ?fi is selected from r2 and r3 and the one closest to the wumpus to be killed is selected. The constraints on delegating agents to roles also are specified. To fill r2 and r3, agents must have arrows, and r2 and r3 must be filled by different agents.

```
(position sniffer (move sense selectTarget))
(position fighter (move shoot))
(plan scanandkill ()
  (Roleteam ((role (r1) sniffer) (role (r2 r3) fighter)))
  (Constraint (hasarrow r2) (hasarrow r3) (NE r2 r3))
  (Process
     (par
        (while (cond (eq 1 1))
            (seq
               (do r1 (selectTarget))
               (if (cond (targetloc r1 ?X1 ?Y1))
                   (do r1 (senseandmoveto ?X1 ?Y1))
               )
            )
        )
        (while (cond (eq 1 1))
           (if (cond (wumpus ?X2 ?Y2))
               (seq
                   (select ?fi (r2 r3) (closestto ?X2 ?Y2))
                   (do ?fi (movecloseto ?X2 ?Y2))
                   (do ?fi (shoot ?X2 ?Y2))
               )
           )
        )
     )
  )
)
```

IV.3.3 Actions in the Processes of Role-Based Plans

In RoB-MALLET, the associations of temporally ordered actions with roles and role variables (described in Sec IV.2.3 and IV.2.4) are specified in the process of a role-based plan. For this purpose, an action in the process needs to include multiple dimensions of information: the first is about "what to do?" (i.e., what actions); the second is about "who does it?" (i.e., with which role(s) or role variable an action is associated); and the third is about "what is its ordering relationship with other actions?" (i.e., temporal ordering on the actions). As our formalisms of role and role variable can accommodate complex actions as described in Sec. IV.2.5, an action could be a complex action. A fourth dimension of information, "with whom and how to coordinate?", can be included to express a complex action.

"What to do?" is straightforwardly represented by operator name, plan name, or role selection except that evaluations of conditions are indirectly specified by the conditions in flow controls (described in Chapter III), such as IF and WHILE statements. For an operator or plan, "who do(es) it?" is represented by associating a role, role variable, or a set of roles with an operator or plan. In RoB-MALLET, it appears as `(DO rolename iopername)` or `(DO rolelist planname)`. As described in Sec. III.5, for the evaluation of a condition, "who do(es) it?" is currently not represented in RoB-MALLET. We assume that the evaluation of the condition is based on the mutual beliefs of all relevant individuals and one of them is selected to be a coordinator to evaluate it (The details about the evaluation of a condition are available in Sec. III.5 and Sec. IV.5.3).

"What are the ordering relationships with other actions?" is specified by flow control described in Sec. III.4. Basically, sequential flow control specifies sequential ordering relationships between actions while parallel flow control specifies independent ordering relationships among actions. It is important that the role selection for a role variable occur before associating actions with the role variable.

As described in Sec. IV.2.5, there are three kinds of complex actions are defined in our teamwork architecture. The first is coordinated operations that require multiple roles

(or role variables) to coordinate so as to perform their own individual actions. This category includes team operators and JOINT DO's. The second kind of complex actions is plan invocation. The syntax and semantics of team operators, JOINT DO and plans have been described in Chapter III.In the process of a role-based plan, the performers of team operators, JOINT DO's, and plan invocations are roles and role variables . The third kind of complex actions is role selection. We have described the syntax of role variable declaration in Sec. IV.3.2.3. .

In the above ways, the temporally ordered actions associated with roles and role variables are specified in the process of a role-based plan. The actions are associated with roles and role variables by DO constructs. Conditional actions are specified by IF and WHILE constructs. The temporal orders among actions are specified by flow controls. Role selections specify selection scopes and selection constraints.

IV.4 Responsibility

In this section, we will first define a notion of responsibility. We will then present a graphic language of responsibility to capture what a responsibility contains. At last, we will translate the role-based plan into a team responsibility.

IV.4.1 Definitions

A responsibility is an obligation [1, 3] by an agent or a team of agents to perform actions, which forms an intention [22, 59] to do the actions. When an agent *ag* takes the responsibility of a role *r*, the agent forms an intention, denoted by **Intend**(*ag*, *r*, *P*), to execute the actions associated with the role *r*. We define **Intend**(*ag*, *r*, *P*) on Cohen and Levesque's persistent goal (described in Chapter III) as follows:

- agent *ag* has a persistent goal relative to the negation of the termination condition of *P* of having done all the actions associated with role *r* (denoted by **Resp**(*r, P*)) in accordance with the temporal orders on the actions; and
- agent *ag* believes throughout that agent *ag* is doing the actions in **Resp**(*r, P*) until all the actions are done, or agent *ag* believes that the termination condition of *P* is satisfied.

We note that some actions in **Resp**(*r, P*) may be conditional. To have such conditional actions done, their conditions need to be checked, and they are completed if their conditions are true or skipped otherwise.

When a team of agents *T* takes the responsibility of the virtual team *VT* in a role-based plan *P* with a team assignment *TA*, the agents in T form a joint intention (denoted by **JIntend**(*T*, *VT*, *TA*, *P*)) to execute the actions associated with the roles (denoted by **Resp**(*VT*, *P*)). We define **JIntend**(*T*, *VT*, *TA*, *P*) based on Cohen and Levesques' joint persistent goal (described in Chapter III) as follows:

- The agents in *T* have a joint persistent goal, relative to the negation of the termination condition of *P,* of having done all actions in **Resp**(*VT, P*) in accordance with the temporal orders on the actions;
- Until all the agents in *T* mutually believe that all actions in **Resp**(*VT, P*) are done or that the termination condition of *P* is satisfied, the agents in *T* mutually believe that, a) they are doing the actions in **Resp**(*VT, P*) in accordance with the temporal orders on the actions; and b) every agent *ag* in *T* takes the responsibility of the role *r* delegated to the agent *ag* according to *TA*.

In (1), to have the actions in **Resp**(*VT, P*) done, each agent *ag* intends to execute the actions associated with role *r* which is delegated to agent *ag* according to the team assignment *TA*. Moreover, if role *r* is involved in a role selection for a role variable *?rv* and role *r* is selected to fill role variable *?rv*, then agent ag intends to execute the actions associated with role variable *?rv*. This can be characterized by:

Similarly, some actions in **Resp***(VT, P)* may be conditional. To have such conditional actions done, their conditions need to be checked, and they are completed if their conditions are true or skipped otherwise.

However, a responsibility is not equivalent to an intention; rather, it causes an agent (or agents) to form an intention by obligating it (or them) to complete actions.

IV.4.2 Representation of Responsibility

To represent the actions and their temporal ordering in a responsibility (either an individual or team responsibility) as well as capturing the dynamicity of responsibility, a graphical language has been developed. In this language, it is straightforward to visualize the actions and their temporal orders in a responsibility. Moreover, a representation in this language will facilitate translating the process of a role-based plan into a graph of a team responsibility (described later in Sec. IV.4.3), decomposing the graph of the team responsibility to graphs of individual responsibilities (described later in Sec. IV.5), and further dynamically composing process knowledge in mental models of agents (described later in Chapter V).

Responsibility is represented as a graph $G(V, E)$. G is directed bipartite graph, where V is a set of nodes and E is a set of directed links connecting nodes. In the graphical representation of responsibility, the nodes represent the entrance and exit of a plan P, the evaluation of conditions, operations, cooperation among roles, and flow control. To distinguish them, different types of nodes are used as the following:

- An operation node representing the invocation of operators by a role or role variable. The information relevant to the operation invocation is attached, including operator name, arguments, and doer (i.e., who perform the operation);
- A condition node representing the evaluation of a condition. The condition is attached;
- A coordination node representing coordination with operations of other roles. In our teamwork architecture, coordination nodes are used to represent coordination specified by team operators and JOINT DO's. The coordination type, AND, OR, or XOR, is attached;
- A plan node representing the invocation of a sub-plan by a sub-team formed from the virtual team responsible for the plan in which the invocation appears. The information relevant to the sub-plan invocation is attached, including sub-plan name, arguments, and the sub-team. To express a sub-plan invocation, every role

in the sub-team has a plan node. The plan node represents the sub-actions in the plan will be associated to the role;

- A selection node representing coordination of a role with other involved roles to select a role to fill a role variable. The information relevant to the selection is attached, including the name of role variable, selection scope and selection constraint. Every role in the selection scope has a selection node;
- A goal node representing the formation of a sub-team to achieve a goal. The sub-team and the goal are attached;
- A connector node expressing flow control. For example, a connector node is used to express the starting or end points of par constructs. Special types of connector nodes are also allowed. An entrance node represents a starting point of the plan *P*. An exit node represents the end points of the plan *P*. A BranchEnd node represents the end of the branches of a branch construct. And a Loopback node represents a loop back to the evaluation of the condition of a loop construct.

We note that for a complex action (a coordinated operation, plan invocation, role selection, or goal achievement), we use a corresponding special node (a coordination, sub-plan, selection, or goal node respectively) for each individual (role or role variable) to represent the sub-actions breaking form the complex actions which are associated with the individual.

The links represent various kinds of relations (described below) among nodes, including dependencies between two nodes, coordination connections, sub-plan connections, goal connections, and connections among roles for the selection of role variables. To represent the relationships with responsibilities of other roles, links are allowed to point to nodes in the responsibility graphs of other roles. Different types of links are used as the following:

- A dependency link representing temporal order of connected nodes. If a dependency link connects two nodes in two responsibility graphs (for different roles or role variables in a role-based plan), it crosses these two graphs. For a team responsibility, a dependency link cannot go outside to another responsibility

graph because the actions connected by the dependency link are in the team responsibility. After a team responsibility is decomposed to individual responsibilities as described later in Sec. IV.5, two internal nodes connected by a dependency link may belong to two individual responsibilities (for roles or role variables), and then the dependency link crosses the graphs of the two individual responsibilities. A loopback link is a special dependency link. Besides temporal order between connected nodes, it also means the link points back to a node where a loop starts. Also, a guard link introduced in next item is a special type of dependency link.

- A guard link representing conditional temporal order of connected nodes. A guard link contains a logic guard (Yes/No). It may stem from a condition node. The logic guard is true if the condition of is true. Also, it may stem from a selection node. The logic guard is true if the role satisfies the selection constraint and the role is selected. It is possible that multiple roles satisfy the selection constraint. However, only the role, which is selected, sets a true value for its logic guard and then proceeds to execute the actions connected by the logic guard. Later in Sec. IV.4.3, we use two guard links with logic guards Yes and No to represent the evaluation of a condition in a flow control, such as a branch or loop construct. If the condition is true, then the actions connected by Yes guard will be executed; otherwise, the actions connected by No guard will be executed;
- A coordination link connecting two coordination nodes with regard to the same coordination. The direction of the link means the communication from the source node of the link to the target node of the link for negotiating the coordination Later in Sec. IV.4.3.6 and Sec. IV.4.3.7, we used two coordination links to connect two coordination nodes in a pair of roles (or role variables). These two coordination links have different communication directions;
- A plan link connecting two plan nodes with regard to the same sub-plan invocation. As stated early, we use a plan node for each role in a role-based plan to represent an invocation of the role-based plan, and a plan node for a role

represents the sub-actions in the role-based plan which will be associated with the role. The direction of the link means the communication from the source node of the link to the target node of the link to notify the target node that the source node is ready to start the sub-plan and ask the target node to start the sub-plan too. Later in Sec. IV.4.3.6, we used two plan links to connect two plan nodes in a pair of roles. These two plan links have different communication directions;

- A selection link connecting two selection nodes with regard to the same role selection. The direction of the link means the communication from the source node of the link to the target node of the link for negotiating which role will be select. Later in Sec. IV.4.3.8, we used two selection links to connect two selection nodes in a pair of roles. These two selection links have different communication directions;
- A goal link connecting two goal nodes with regard to a same goal achievement. The direction of the link means the communication from the source node of the link to the target node of the link for negotiating a course of actions to achieve the goal. Later in Sec. IV.4.3.9, we used two goal links to connect two goal nodes in a pair of roles. These two goal links have different communication directions

The graphic language contains various components to represent complex actions. For example, coordination, sub-plan, goal, and selection nodes and links are used in the graphical language to express various kinds of complex actions. If additional constructs were added into RoB-MALLET to express different kinds of complex actions, more types of nodes and links could be added to the graphical language. Thus, the graphical representation can easily be extended.

For example, Figure 11 shown in Sec. IV.4.3.10 shows a team responsibility of the virtual team in a role based plan `scanandcollect` described in IV.3.2.6. Figure 18 shown in Sec. IV.5.5 shows the responsibilities of roles r_1 and r_2, which are decomposed from the team responsibility shown in Figure 11 shown in Sec. IV.4.3.10.

IV.4.3 Translating a Role-Based Plan to a Team Responsibility Graph

As described in Sec. IV.4, the team responsibility of a virtual team captures the temporally ordered actions associated with the virtual team and the impact on the mental state of the agents filling the virtual team. By forming a joint intention, i.e., **JIntend***(T, VT, TA, P)*, the agents can execute the actions contained in the team responsibility as a team effort. In our teamwork architecture, a team task is specified by a team of agents invoking a role-based plan in RoB-MALLET. In order to enable the agents to execute the role-based plan, we translate the process of the role-based plan into a team responsibility graph of the virtual team in the role-based plan.

As listed in APPENDIX A, the syntax of a plan process in RoB-MALLET is defined in a recursive way. The process of a role-based plan itself is a *MalletProcess*. A *MalletProcess* may be one of various types of process constructs, including sequential (*SeqMalletProcess*), parallel (*ParMalletProcess*), branch (*BranchMalletProcess*), loop (*LoopMalletProcess*), joint do (*JoinDoMalletProcess*), role selection (*SelectMalletProcess*), do (*DoMalletProcess*), and achieve (*GoalMalletProcess*) constructs. Moreover, these constructs are recursively defined by *MalletProcess*es.

Following the recursive definition of the process of a role-based plan, we present a set of recursively applied algorithms to translate the process of a role-based plan into a team responsibility graph for the virtual team in the role-based plan in a recursive way.

- An algorithm, *GenerateTeamResponsibility (P)*, is used to translate the process of a role-based plan *P* into a team responsibility. An entrance node *StartNode* and an exit node *ExitNode* are created to represent the starting and finishing point of the process respectively. It basically calls *Translate_MalletProcess (MalletProcess, StartNode, ExitNode, Resp)* to handle the *MalletProcess* of the process.
- An algorithm, *Translate_MalletProcess(MalletProcess, Pnode, Snode, Resp)*, is used to translate a *MalletProcess* into responsibility components. *MalletProcess* is the *MalletProcess* to be translated. *Pnode* is a responsibility node corresponding to an action or a control flow (i.e., a connector node) which occurs

before the *MalletProcess*. *Snode* is a responsibility node corresponding to an action or a control flow which occurs after the *MalletProcess*;

- A set of algorithms are used to handle different types of *MalletProcess*es, including sequential, parallel, branch, loop, joint do, selection, do, and achieve constructs. Because these constructs are recursively defined by *MalletProcess*es, the algorithms call *Translate_MalletProcess* described above to handle the *MalletProcess*es in these constructs.

A responsibility contains a set of nodes *Nodes* and a set of links *Links*. Each node or link in the responsibility may attach additional information according to its type as described in Sec. IV.4.2. Initially, the sets *Nodes* and *Links* are empty. In the following algorithms, every newly created node is added to set *Nodes* of the responsibility in the algorithms; and every new created link is added to set *Links*. Moreover, each newly created node or link is given a unique name.

In the next sections, we will describe the algorithms for translating a role-based plan to a team responsibility graph. Also, for easier understanding, we will use figures to illustrate how to translate different types of constructs into responsibility representations. In those figures, we use solid blocks and solid arrows to represent nodes and links created by the algorithms; we used dotted blocks to represent the *MalletProcess*es in the constructs being translated; and we use dotted arrows to represent the links to be added for connecting the nodes in current team responsibility graph and the nodes added in the future when translating the *MalletProcess*es in the constructs.

IV.4.3.1 MalletProcess

A *MalletProcess* to be translated may be a sequential, parallel, branch, loop, joint do, role selection, do, or achieve construct. To translate a *MalletProcess*, an algorithm, *Translate_MalletProcess*, checks the type of the *MalletProcess* and then invokes corresponding algorithms (described in later sections) to handle the *MalletProcess* as follows:

Algorithm 1. Translating a MalletProcess into responsibility

```
Translate_MalletProcess(MalletProcess, Pnode, Snode, Resp)
  Begin
    Case MalletProcess of
      Sequential construct:
        Translate_SeqMalletProcess(MalletProcess, Pnode, Snode, Resp);
      Parallel construct:
        Translate_ParMalletProcess(MalletProcess, Pnode, Snode, Resp);
      Branch construct:
        Translate_BranchMalletProcess(MalletProcess, Pnode, Snode, Resp);
      Loop construct:
        Translate_LoopMalletProcess(MalletProcess, Pnode, Snode, Resp);
      Do construct:
        Translate_DoMalletProcess(MalletProcess, Pnode, Snode, Resp);
      Joint do construct:
        Translate_JointDoMalletProcess(MalletProcess, Pnode, Snode, Resp);
      Selection construct:
        Translate_SeletionMalletProcess(MalletProcess, Pnode, Snode, Resp);
      Achieve construct:
        Translate_GoalMalletProcess(MalletProcess, Pnode, Snode, Resp);
    Endcase;
  End
```

IV.4.3.2 Sequential Construct

A sequential construct contains a list of *MalletProcess*, which are executed in sequential order. The following is the syntax of sequential construct in MALLET:

SeqMalletProcess ::= "(" "SEQ" MalletProcess+ ")"

Figure 1 illustrates how a sequential construct is translated into a responsibility representation. The detail about this translation is given by Algorithm 2.

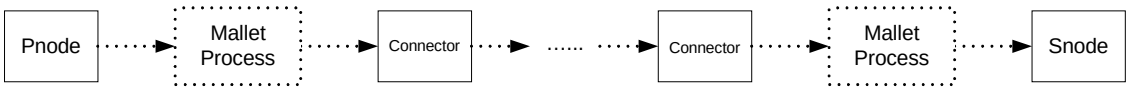

Figure 1. Responsibility representation of a sequential construct.

Algorithm 2. Translating a sequential construct into responsibility

Translate_SeqMalletProcess (*SeqMalletProcess*, *Pnode*, *Snode*, *Resp*)

Begin

Predecessor = *Pnode*;

For each *MalletProcess* in the list of *SeqMalletProcess* do

If *MalletProcess* is not last one, then

Create a connector node *n* in *Resp*;

Successor = *n*;

Else

Successor = *Snode*;

Endif;

Translate_MalletProcess(*MalletProcess*, *Predecessor*, *Successor*, *Resp*);

Endfor;

End

IV.4.3.3 Parallel Construct

A parallel construct contains a list of *MalletProcess*, which are executed in parallel. The following is the syntax of parallel construct in MALLET:

ParMalletProcess ::= "(" "PAR" MalletProcess+ ")"

Figure 2 illustrates how a parallel construct is translated into a responsibility representation. The detail about this translation is given by Algorithm 3.

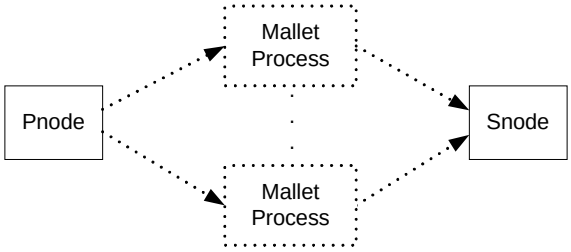

Figure 2. Responsibility representation of a parallel construct.

Algorithm 3. Translating a parallel construct into responsibility

Translate_ParMalletProcess (*ParMalletProcess*, *Pnode*, *Snode*, *Resp*)

Begin

For each *MalletProcess* in the list of *ParMalletProcess* do

Translate_MalletProcess(*MalletProcess*, *Pnode*, *Snode*, *Resp*);

Endfor;

End

IV.4.3.4 Branch Construct

A branch construct contains a condition and one or two *MalletProcess*es. If the condition is true, the first *MalletProcess* is executed; otherwise, the second is executed if there is one. The following is the syntax of branch construct in MALLET:

BranchMalletProcess ::= "(" "IF" "(" "COND" Pred+ ")" MalletProcess MalletProcess? ")"

Figure 3 illustrates how a branch is translated into a responsibility representation. The detail about this translation is given by Algorithm 4, as shown in Figure 3. The condition in the branch construct (i.e., *Pred+* in the above *BranchMalletProcess*) are attached to *Cnode* below in Figure 3 statically. The condition may contain some variables. When the branch construct is executed, the variables are dynamically substituted by the values for these variables.

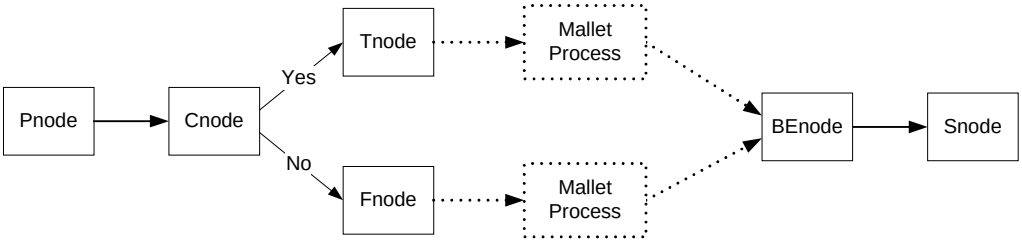

Figure 3. Responsibility representation of a branch construct.

Algorithm 4. Translating a branch construct into responsibility

Translate_BranchMalletProcess (*BranchMalletProcess*, *Pnode*, *Snode*, *Resp*)

Begin

Create a condition node *Cnode* in *Resp*;

Add the condition of *BranchMalletProcess* to *Cnode*;

Create a dependency link *Slink* in *Resp*;

Connect *Pnode* and *Cnode* by link *Slink* (from *Pnode* to *Cnode*);

Create a branch end connector node *BEnode* in *Resp*;

Create a dependency link *Elink* in *Resp*;

Connect *BEnode* and *Snode* by link *Elink* (from *BEnode* to *Snode*);

Create a connector node *Tnode* in *Resp*;

Create a guard link *Tlink* in *Resp* and label *Tlink* as "Yes";

Connect *Cnode* and *Tnode* by link *Tlink* (from *Cnode* to *Tnode*);

Let the first *MalletProcess* of *BranchMalletProcess* be *MalletProcess1*;

Translate_MalletProcess(*MalletProcess1*, *Tnode*, *BEnode*, *Resp*);

Create a connector node *Fnode* in *Resp*;

Create a guard link *Flink* in *Resp* and label *Flink* as "No";

Connect *Cnode* and *Fnode* by link *Flink* (from *Cnode* to *Fnode*);

If the second *MalletProcess* of *BranchMalletProcess* is present, then

Let the second *MalletProcess* of *BranchMalletProcess* be *MalletProcess2*;

Translate_MalletProcess(*MalletProcess2*, *Fnode*, *BEnode*, *Resp*);

Else

Create a dependency link *FElink* in *Resp*;

Connect *Fnode* and *BEnode* by link *FElink* (from *Fnode* to *BEnode*);

Endif;

End

IV.4.3.5 Loop Construct

A loop construct contains a condition and a *MalletProcess*. If the condition is true, the *MalletProcess* is executed and then the condition is evaluated again. This procedure repeats until the condition is false. The following is the syntax of loop construct in MALLET:

LoopMalletProcess ::= "(" "WHILE" "(" "COND" Pred+ ")" MalletProcess ")"

Figure 4 illustrates how a loop branch is translated into a responsibility representation. The detail about this translation is given by Algorithm 5. The condition in the branch construct (i.e., *Pred+* in the above *LoopMalletProcess*) is attached to Cnode below in Figure 4 statically. The condition may contain some variables. When the loop construct is executed, the variables are dynamically substituted by the values for these variables.

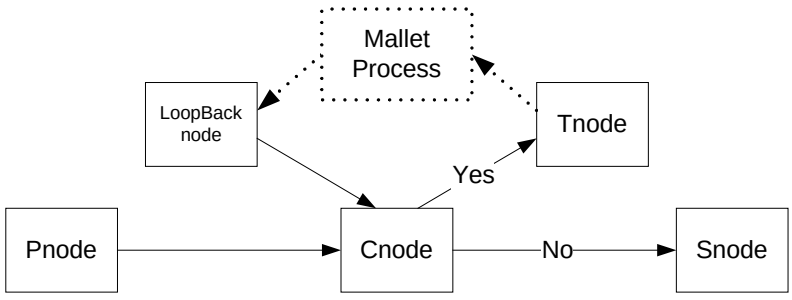

Figure 4. Responsibility representation of a loop construct.

Algorithm 5. Translating a loop construct into responsibility

Translate_LoopMalletProcess (*LoopMalletProcess*, *Pnode*, *Snode, Resp*)

Begin

Create a condition node *Cnode* in *Resp*;

Add the condition of *LoopMalletProcess* to *Cnode*;

Create a dependency link *Slink* in *Resp*;

Connect *Pnode* and *Cnode* by link *Slink* (from *Pnode* to *Cnode*);

Create a connector node *Tnode* in *Resp*;

Create a guard link *Tlink* in *Resp* and label *Tlink* as "Yes";

Connect *Cnode* and *Tnode* by link *Tlink* (from *Cnode* to *Tnode*);

Create a connector node *LoopBacknode* in *Resp*;

Create a loop back dependency link *LoopBacklink* in *Resp*;

Connect *LoopBacknode* and *Tnode* by link *LoopBacklink* (from *LoopBacknode* to *Tnode*);

Translate_MalletProcess(*MalletProcess*, *Tnode*, *LoopBacknode*, *Resp*);

Create a guard link *Flink* in *Resp* and label *Flink* as "No";

Connect *Cnode* and *Fnode* by link *Flink* (from *Cnode* to *Snode*);

End

IV.4.3.6 Do Construct

A do construct is to invoke an operator or plan by a role, role variable, or a set of roles. The operator may be either team or individual operator. If it is an individual operator, the doer is a role or role variable; otherwise, the doer is a list of roles. The following is the syntax of do construct in MALLET:

DoMalletProcess ::= "(" "DO" ByWhomSpec Invocation ")"

Invocation ::= "(" PlanOrOperName (<IDENTIFIER> | <VARIABLE>) ")"*

ByWhomSpec ::= RoleName | RoleVariableName | RoleList

Figure 5, Figure 6 and Figure 7 illustrate how a do construct is translated into the representation in the graphical language of responsibility. The details about these translations are given by Algorithm 6.

When processing a do construct, some nodes are naturally ownerized (We use the term "ownerize" as "set owner", for example, "ownerize node *n* to role *r*" means "set r as the be owner of node n". If a role or role variable is ownerized to a node, that node is responsible to perform the action corresponding to the node.) to a role or role variable as follows:

1. an operator node is ownerized to the performer of an operator invocation. For example, `(DO r1 (op1))`. An operator node is created for operator `op1` and it is ownerized to role `r1`;
2. a plan node is ownerized to one of the performers of a sub-plan invocation. For example, `(DO (r1 r2) (plan1))`. For each performer of `plan1`, `r1` or `r2`, a plan node is created and ownerized correspondingly. Each plan node represents the sub-actions in plan `plan1` which will be associated to the role to which the plan node is ownerized;
3. a coordination node is ownerized to one of the performers of a team operator invocation. For example, `(DO (r1 r2) (top1))`, where `top1` is a team operator. For each performer of `top1`, `r1` or `r2`, a coordination node and an operator node are created and ownerized correspondingly. A coordination node of a role represents a routine by which the role coordinates with others to make the decision on the coordination.

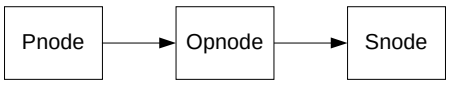

Figure 5. Responsibility representation of a do individual operator construct.

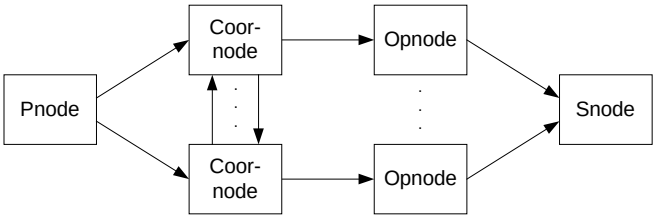

Figure 6. Responsibility representation of a do team operator construct.

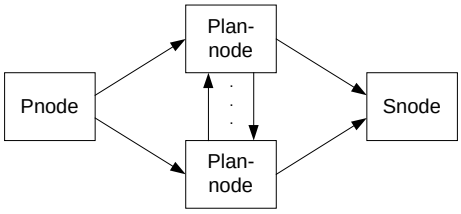

Figure 7. Responsibility representation of a do plan construct.

Algorithm 6. Translating a do construct into responsibility

Translate_DoMalletProcess (*DoMalletProcess*, *Pnode*, *Snode*, *Resp*)

Begin

Create a doer set *Doers*;

Add the roles and role variables in *ByWhomSpec* of *DoMalletProcess* to *Doers*;

Case *Invocation* of

Individual operator:

Create an operator node *Opnode* in *Resp*;

Set the arguments of *Opnode* as the identifiers or variables in *Invocation*;

Set the only role or role variable in *Doers* as the owner of *Opnode*;

Create a dependency link *Plink* in *Resp*;

Connect *Pnode* and *Opnode* by link *Plink* (from *Pnode* to *Opnode*);

Create a dependency link *Slink* in *Resp*;

Connect *Opnode* and *Snode* by link *Slink* (from *Opnode* to *Snode*);

Team operator:

For each *doer* in *Doers* do

Create a coordination node *Coornode* in *Resp*;

Set *doer* as the owner of *Coornode*;

Set the coordination type of *Coornode* as the coordination type of the team operator;

Create a dependency link *Plink* in *Resp*;

Connect *Pnode* and *Coornode* by link *Plink* (from *Pnode* to *Coornode*);

Create an operator node *Opnode* in *Resp*;

Set the arguments of *Opnode* as the identifiers or variables in *Invocation*;

Set *doer* as the owner of *Opnode*;

Create a dependency link *Mlink* in *Resp*;

Connect *Coornode* and *Opnode* by link *Mlink* (from *Coornode* to *Opnode*);

Create a dependency link *Slink* in *Resp*;

Connect *Opnode* and *Snode* by link *Slink* (from *Opnode* to *Snode*);

Endfor;

For each *doer1* in *Doers* do

Let *doer1*'s coordination node be *Coornode1*;

For each *doer2* in *Doers* except *doer1* do

Let *doer2*'s coordination node be *Coornode2*;

Create a coordination link *Coorlink* in *Resp*;

Connect *Coornode1*and *Coornode2* by link *Coorlink* (from *Coornode1* to *Coornode2*);

Endfor;

Plan:

For each *doer* in *Doers* do

Create a plan node *Plannode* in *Resp*;

Set the arguments of *Plannode* as the identifiers or variables in *Invocation*;

Set the performers of *Plannode* as *Doers*;

Set *doer* as the owner of *Plannode*;

Create a dependency link *Plink* in *Resp*;

Connect *Pnode* and *Plannode* by link *Plink* (from *Pnode* to *Plannode*);

Create a dependency link *Slink* in *Resp*;

Connect *Plannode* and *Snode* by link *Slink* (from *Plannode* to *Snode*);
Endfor;

For each *doer1* in *Doers* do
Let *doer1*'s plan node be *Plannode1*;
For each *doer2* in *Doers* except *doer1* do
Let *doer2*'s plan node be *Plannode2*;
Create a plan link *Planlink* in *Resp*;
Connect *Plannode1* and *Plannode2* by link *Planlink* (from *Plannode1* to *Plannode2*);
EndFor;
Endfor;
Endcase;
End

IV.4.3.7 Joint Do Construct

The purpose of a JOINT DO construct is to coordinate a set of roles and role variables during their execution of a list of *MalletProcesses*. The coordination type among these *MalletProcess* can be AND, OR and XOR. The following is the syntax of JOINT DO construct in MALLET:

JoinDoMalletProcess ::= "(" "JOINTDO" ("AND" | "OR" | "XOR")? ("(" ByWhomSpec MalletProcess ")")+ ")"

Figure 8 illustrates how a JOINT DO construct is translated a responsibility representation. The detail about this translation is given by Algorithm 7.

The coordination nodes for a JOINT DO construct are slightly different from those of a do construct of a team operator. The coordination nodes for a JOINT DO are to coordinate the branches of *MalletProcess* in a JOINT DO construct. Every *MalletProcess* branch could contain a process of actions to be performed by multiple roles and role variables. Thus, the coordination nodes for a joint do cannot be ownerized to some roles or role variables in Algorithm 7. Later in Sec. IV.5, we will show how

such a coordination node is ownerized to some role or role variable for decomposing a team responsibility to individual responsibilities.

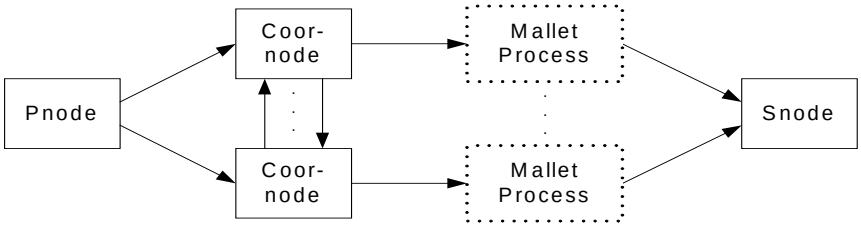

Figure 8. Responsibility representation of a joint do construct.

Algorithm 7. Translating a joint do construct into responsibility

Translate_JointDoMalletProcess(*JoinDoMalletProcess*, *Pnode*, *Snode*, *Resp*)

Begin

For each *MalletProcess* in *JoinDoMalletProcess* do

Create a coordination node *Coornode* in *Resp*;

Set the coordination type of *Coornode* as the coordination type of the *JoinDoMalletProcess*;

Create a dependency link *Plink* in *Resp*;

Connect *Pnode* and *Coornode* by link *Plink* (from *Pnode* to *Coornode*);

Translate_MalletProcess(*MalletProcess*, *Coornode*, *Snode*, *Resp*);

Endfor;

For each *MalletProcess1* in *JoinDoMalletProcess* do

Let *MalletProcess1*'s coordination node be *Coornode1*;

For each *MalletProcess2* in *JoinDoMalletProcess* except *MalletProcess1* do

Let *MalletProcess2*'s coordination node be *Coornode2*;

Create a coordination link *Coorlink* in *Resp*;

Connect *Coornode1* and *Coornode2* by link *Coorlink* (from *Coornode1* to *Coornode2*);

Endfor;

Endfor;

End

IV.4.3.8 Selection Construct

A selection construct is used to specify selection of a role from a list of roles to fill a role variable so that the role performs the actions associated with the role variable. A list of selection constraints functions as the criteria for selecting a role from the selection scope. The following is the syntax of selection construct in MALLET:

SelectMalletProcess ::= (SELECT RoleVariableName RoleList ConstraintsList)

Figure 9 illustrates how a selection construct is translated into a responsibility representation. The detail about this translation is given by Algorithm 8.

When processing a selection construct, a selection node is created for each involved role and is naturally ownerized to the role correspondingly.

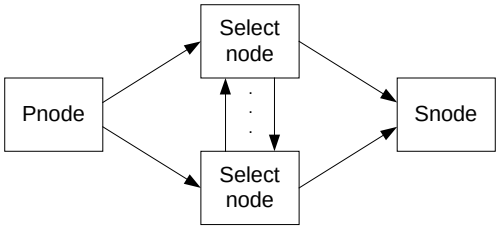

Figure 9. Responsibility representation of a selection construct.

Algorithm 8. Translating a selection construct into responsibility

Translate_SeletionMalletProcess (*SelectMalletProcess*, *Pnode*, *Snode*, *Resp*)

Begin

For each *role* in *RoleList* of *SelectMalletProcess* do

Create a selection node *Selectnode* in *Resp*;

Set selection constraints of *Selectnode* as *ConstraintsList* of *SelectMalletProcess*;

Set *role* as the owner of *Selectnode*;

Create a dependency link *Plink* in *Resp*;

Connect *Pnode* and *Selectnode* by link *Plink* (from *Pnode* to *Selectnode*);

Create a dependency link *Slink* in *Resp*;

Connect *Selectnode* and *Snode* by link *Slink* (from *Selectnode* to *Snode*);

Endfor;

For each *role1* in *RoleList* of *SelectMalletProcess* do

Let *role1*'s selection node be *Selectnode1*;

For each *role2* in *RoleList* of *SelectMalletProcess* except *role1* do

Let *role2*'s coordination node be *Selectnode2*;

Create a selection link *Selectlink* in *Resp*;

Connect *Selectnode1* and *Selectnode2* by link *Selectlink* (from *Selectnode1* to *Selectnode2*);

Endfor;

Endfor;

End

IV.4.3.9 Achieve Construct

A achieve construct is used to specify that a set of roles is to achieve a goal without giving specific action (either operator or plan). The goal is a conjunction of predicates. The following is the syntax of achieve construct in MALLET:

GoalMalletProcess ::= "(" "ACHIEVE" ByWhomSpec Pred+ ")"

Figure 10 illustrates how a achieve construct is translated into a responsibility representation. The detail about this translation is given by Algorithm 9.

When processing an achieve construct, a goal node is created for each involved role and is naturally ownerized to the role correspondingly.

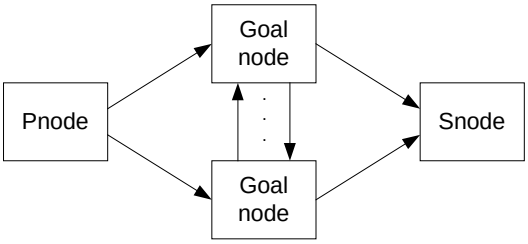

Figure 10. Responsibility representation of an achieve construct.

Algorithm 9. Translating an achieve construct into responsibility

Translate_GoalMalletProcess (*GoalMalletProcess*, *Pnode*, *Snode*, *Resp*)

Begin

For each *role* in *ByWhomSpec* of *GoalMalletProcess* do

Create a goal node *Goalnode* in *Resp*;

Set the goal of *Goalnode* as the conjunction of *Preds* of *GoalMalletProcess*;

Set *role* as the owner of *Goalnode*;

Create a dependency link *Plink* in *Resp*;

Connect *Pnode* and *Goalnode* by link *Plink* (from *Pnode* to *Goalnode*);

Create a dependency link *Slink* in *Resp*;

Connect *Goalnode* and *Snode* by link *Slink* (from *Goalnode* to *Snode*);

Endfor;

For each *role1* in *ByWhomSpec* of *GoalMalletProcess* do

Let *role1*'s selection node be *Goalnode1*;

For each *role2* in *ByWhomSpec* of *GoalMalletProcess* except *role1* do

Let *role2*'s coordination node be *Goalnode2*;

Create a goal link *Goallink* in *Resp*;

Connect *Goalnode1* and *Goalnode2*by link *Goallink* (from *Goalnode1* to *Goalnode2*);

Endfor;

Endfor;

End

IV.4.3.10 Generating A Team Responsibility Graph

In previous sections, we have described how various MALLET constructs are translated into responsibility representations. The process of a role-based plan is a *MalletProcess*, which is recursively constructed by them. The algorithm for generating team responsibility utilizes the MALLET parser to build the syntax tree for a role-based plan so as to recognize the constructs in the process, and then uses the above algorithms

to handle various constructs in the plan. The following is the algorithm of generating team responsibility graph:

Algorithm 10. Generating a team responsibility graph for a role-based plan *P*

GenerateTeamResponsibility (*P*)

Begin

Create a team responsibility, *Resp*;

Build the syntax tree of *P*, *tree*, by the MALLET parser;

Let *MalletProcess* be the root of *tree*;

Create an entrance node, *StartNode*, in *Resp*;

Create an exit node, *ExitNode*, in *Resp*;

Translate_MalletProcess(*MalletProcess*, *StartNode*, *ExitNode*, *Resp*);

End

The team responsibility graph of plan `scanandcollect` described in Sec. IV.3.2.6 generated by Algorithm 10 is shown in Figure 11.

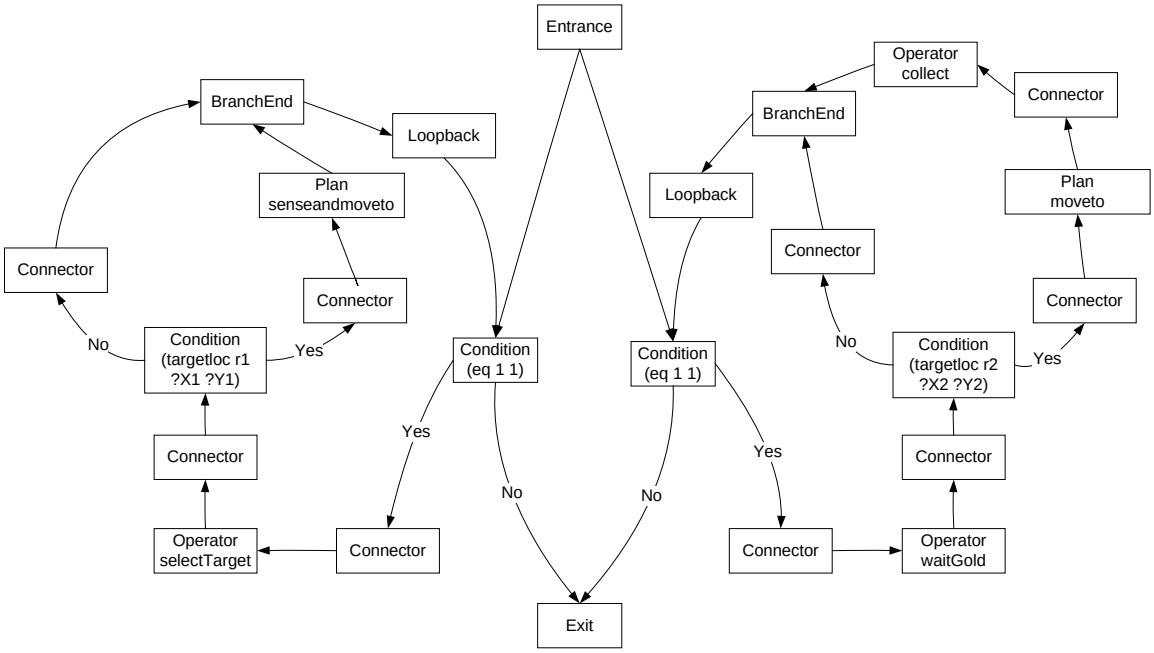

Figure 11. Team responsibility of plan `scanandcollect`.

We note that redundant connector nodes may be used for easy translations the algorithms described in previous sections. The redundant connector nodes can be reduced. The details about how to reduce redundant connector nodes are given in APPENDIX A. After a role-based plan is translated into a team responsibility graph by Algorithm 10, the strategies described in APPENDIX A are applied to reduce redundant connector nodes. The team responsibility graph of plan `scanandcollect` after reducing redundant connector nodes is shown in Figure 12.

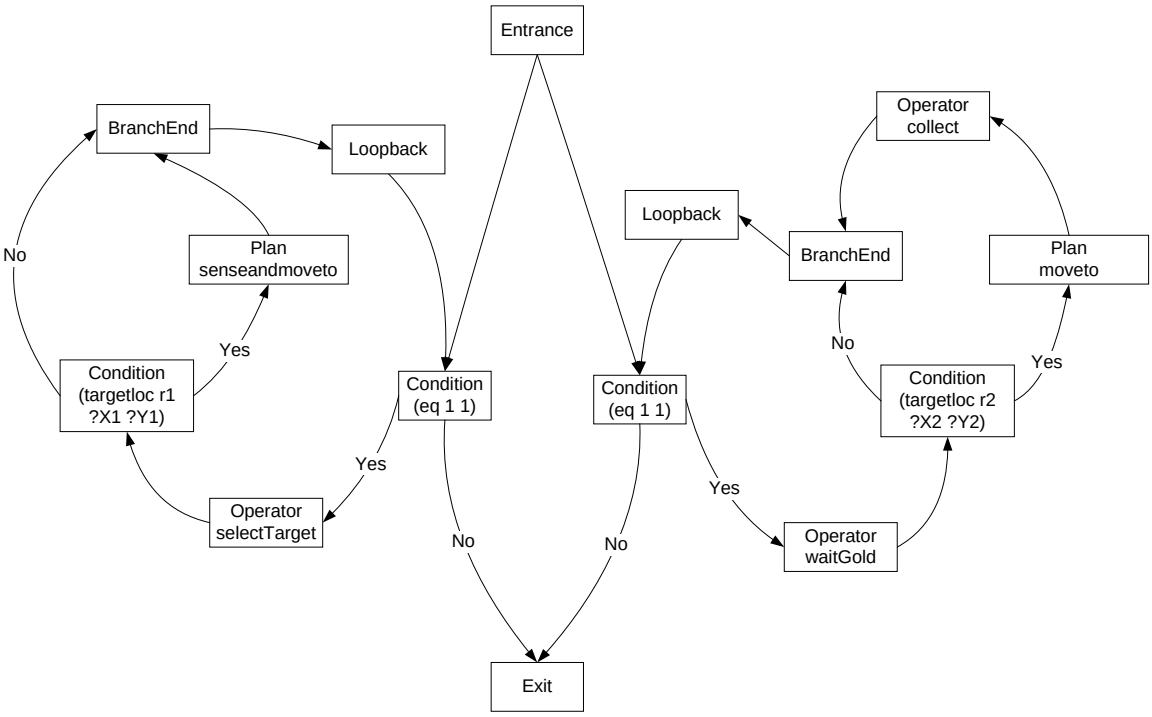

Figure 12. Team responsibility of plan `scanandcollect` after reduction.

IV.5 Decomposing a Team Responsibility Graph to Individual Responsibility Graphs

As described in Sec. IV.4.1, when a team of agents *T* takes the team responsibility of the virtual team *VT* in a role-based plan *P*, the agents in *T* form a joint intention to perform the actions in the team responsibility according to the team assignment *TA* from *VT* to *T*, i.e., **JIntend***(T, VT, TA, P)*. Moreover, as described in condition (1) of

JIntend*(T, VT, TA, P)*, in order to have the actions in the team responsibility done, each agent *ag* in *T* takes the responsibility of the role *r* delegated to the agent according to the team assignment *TA*, i.e., **Intend***(ag, r, P)*. In order to enable agents to execute a team responsibility by each of the agents executing the individual responsibility of the role delegated to the agent, we decompose the team responsibility graph to individual responsibility graphs. The individual responsibilities taken by agents are the responsibilities of the roles in the virtual team *VT*; however, as explained in Sec. IV.4.1, the responsibilities of the role variables in the role-based plan *P* are dynamically included in the responsibilities of the roles. For this reason, we decompose the team responsibility into the individual responsibilities of all roles and role variables in the role-based plan P. During the execution of the role-based plan *P* each agent dynamically uses the individual responsibilities of role variables to expand the responsibilities of the role delegated to the agent.

In this section, we will present a method to decompose a team responsibility graph into individual responsibility graphs. In general, we decompose a team responsibility graph into individual responsibility graphs by ownerizing the nodes and links in the team responsibility graph to the roles and role variables in the role-based plan. As a result, the individual responsibility graph of a role (or role variable) contains the nodes and links ownerized to the role (or role variable). We will first explain this general idea and then describe our ways to handle various unownerized nodes. At last, we will give an algorithm of decomposing a team responsibility graph to individual responsibility graphs through our ownerization methods.

IV.5.1 Decomposing a Team Responsibility through Ownerizations

According to the definition of individual responsibility described in Sec. IV.4.1, the decomposition of a team responsibility to individual responsibilities needs to ensure that every action in the team responsibility (i.e., the actions in the process of the role-based plan corresponding to the team responsibility) is included into the responsibility of the role (or role variable) with which the action is associated. Moreover, according to the definition of team responsibility described in Sec. IV.4.1, the decomposition of a team

responsibility graph to individual responsibility graph also needs to ensure that the temporal orders on the actions in the team responsibility can be preserved so that the joint intention to the actions in the team responsibility can be realized by individual intentions to the actions in the individual responsibilities as described by **JIntend***(T, VT, TA, P)* in Sec. IV.4.1.

As described in Sec. IV.4.3, when we translate the process of a role-based plan into a team responsibility graph, we translate the actions in the process into corresponding nodes and ownerize the nodes to the roles or role variables with which the actions are associated. If an action is an invocation of an individual operator, the action is translated into an operator node and the operator node is ownerized to the role (or role variable) which invokes the operator. If an action is an invocation of a team operator, the actions are translated into individual operators corresponding to the team operator and one of the individual operators is ownerized to each involved role (or role variable). If an action is a complex action other than a team operator, such as a role selection, a sub-plan invocation, or a goal achievement, the action is translated to a set of nodes corresponding to the action and one of the nodes is ownerized to each involved role (or role variable).

However, other components in the team responsibility graph are ownerized to roles and role variables, including connector nodes, condition nodes, coordination nodes for joint do's, and links. All these components require corresponding processing as described in Sec. IV.4.2. Moreover, these components are critical for determining whether to perform the actions, which have been ownerized to individual responsibilities as described above, and ensuring the temporal orders on the actions. For example, a connector after the nodes corresponding the actions in a PAR construct represents the rendezvous of all branches. A condition node corresponding to an IF construct represents the evaluation of the condition in the IF construct which decide if the actions in the IF construct will be executed. Thus, all these components also need to be ownerized to either a role (or role variable).

If a role (or role variable) is the owner of a node, the role (or role variable) is responsible for handling the node according to the meaning of the node. For example, if an operator node is ownerized to a role *r*, then the role *r* (eventually the agent to which the r_1 is delegated) is responsible for executing the corresponding operator. If a role *r* is the owner of a link, the role *r* is responsible to handle the node according to the meaning of the node. For example, if a dependency link l_1, which connects a node n_1 to a node n_2, is ownerized to r_1, then r_1 is responsible for notifying the owner of n_2 that n_1 has been handled so that the owner of n_2 knows that n_2 is ready to go. And, if a coordination link l_2, which connects a coordination node n_3 to a coordination node n_4, is onwerized to a role r_2, then the role r_2 is responsible to coordinate the performance of the next node with node n_4.

Because a link connects two nodes and represents the relationship between two nodes, the owner of the link depends on what the owners of the two nodes are. If the owners of the two nodes are the same role or role variable, the owner of the link is consequently the same role or role variable. Otherwise, if the owners of the two nodes are different, the link implies communication between the owners of the two nodes and the source of the link is the initiator of the communication. It is thus very natural to let the owner of the link be that of the source node. For this reason, we focus on how to handle the unownerized nodes described above.

In our teamwork architecture, a philosophy taken to ownerize those nodes is keeping the amount of communication as low as possible. In next sections, we will show how to handle unownerized connector, condition and coordination nodes.

IV.5.2 Ownerizing Unownerized Connector Nodes

If an unownerized node is a connector node, it is ownerized to the owner of one of successor nodes (i.e., the nodes connected by its output links). Semantically, a connector node functions as flow control and represents the starting point of executing its successor nodes. In the following, we show the reason why we adopt this strategy by comparing different choices of ownerizing a connector node:

- Ownerize the connector node to the owners of all its successor nodes. This choice can be understood by an example of lifting a heavy table by multiple people. To lift a heavy table, all people shout "1, 2, 3, lift" and then lift together.
- Ownerize the connector node to the owner of one of its successor nodes. Applied to the above example, one of the people shouts "1, 2, 3, lift" and then all of them lift together. Although selecting different coordinators may have a little impact on the coordination (e.g., the person in the middle is better selected for other to hear his/her voice), we assume that the selection is random. This is done off line before execution so no communication is required to accomplish the selection.
- Or ownerize the connector node to other irrelevant role (or role variable). Applied to the above example, someone else standby shouts "1, 2, 3, lift" and then all of them lift together.

For the above choices which imply coordinators, we assume that all individuals know the coordinator and accept his/her coordination once an individual is selected as the coordinator.

From the perspective of communication, each shout means a broadcast to other people. Suppose that *n* people participate in lift the heavy table, the amounts of communication cost in the above choices are *n*(n-1)*, *n-1*, and *n* respectively. Apparently, the second choice is the most efficient in communication.

Moreover, if the first choice is applied, each individual needs to have a copy of the node. Doing so may cause additional communication. For example, if the connector node is after a node N. After the node N is executed, the owner of N needs to notify all individuals so that they know that their copies of the connector nodes are ready. If the third choice is applied, additional cost may be needed to find a proper irrelevant role.

For the above reasons, we apply the second choice, i.e., ownerizing to the owner of one of its successor nodes. For example, in Figure 13, N1 is an unownerized connector node with successor nodes from N2 to Ni. N2 is ownerized to r2. According to this strategy, N1 can be ownerized to r2 too.

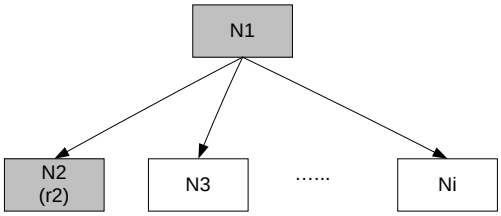

Figure 13. Ownerize an entrance or connector node by one of its successor node.

It is possible that an unownerized connector is an ending point of some node(s), such as the exit node in the team responsibility. If so, the unownerized connector node cannot be handled by the above strategy. To handle such an unownerized connector, we ownerize it to the owner of one of its predecessor nodes (i.e., the nodes connected by its input links). For similar reasons, we choose to ownerize to the owner of one of its predecessor nodes. For example, in Figure 14, N1 is such an unownerized connector node with predecessor nodes from N2 to Ni. N2 is ownerized to r2. According to this strategy, N1 can be ownerized to r2 too.

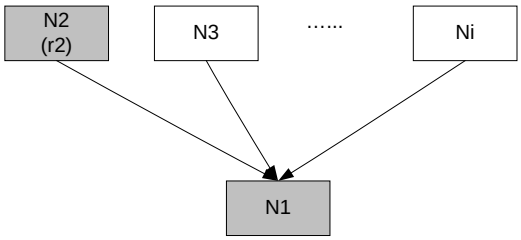

Figure 14. Ownerize an exit or connector node by one of its predecessor node.

IV.5.3 Ownerizing Unownerized Condition Nodes

The algorithms in Sec. IV.4.3 translate the condition in a branch or loop construct to a condition node representing the evaluation of the condition. However, the condition node is not ownerized to any individual(s) so that the individual(s) are responsible to evaluate the condition. As explained in Sec. III.5, in RoB-MALLET, the evaluation of the condition in a flow control is semantically based on the mutual belief on the condition of all relevant individuals. As described in Sec. III.3.2, there are two methods for implementing the evaluation of a condition as a mutual belief: 1) an involved

individual serves as coordinator and evaluates the condition based on mutual beliefs of all individuals; and 2) all individuals separately evaluate the condition based on mutual beliefs of all individuals. For a better communication performance, we choose the first method. One involved individual evaluates the condition based on based on mutual beliefs of all individuals and propagate the result of other involved individuals. Corresponding to this method, an unownerized condition node is ownerized to one of relevant individuals.

As described in Sec. IV.4.3.4 and IV.4.3.5, when a branch or loop construct is translated to a responsibility representation, relevant actions in the construct are translated into corresponding nodes after the condition node corresponding to the condition in the construct. Although additional connector nodes are included between the condition node and the nodes corresponding to the relevant actions, the connector nodes are reduced as described in APPENDIX A. So, the nodes corresponding to the relevant actions become the successor nodes of the condition node. We know that the nodes corresponding to the relevant actions have been ownerized to the relevant individuals when translating the relevant actions to the nodes as described in Sec. IV.4.3. So, the relevant individuals are the owner of these successor nodes of the condition node. However, the successor nodes, which are connected to the condition node through false guard links, may not represent relevant actions. For example, an action sequentially following a loop construct is translated into a successor node that is connected to the condition node corresponding to the condition of the loop construct through a true guard link.

For these reasons, an unownerized condition node is ownerized to the owner of one of successor nodes that are connected to the condition node through true guard links. For example, Figure 15 shows that condition node N1 is ownerized to r1 by ownerizing N1 to the owner of its successor node N2 that is connected by true output link.

In Sec. III.5.3, we proposed a syntax changes for a clear semantics on the evaluation of conditions in flow controls. If those syntax changes are taken, this strategy can be

easily changed to ownerize one of the individuals responsible of evaluating the condition.

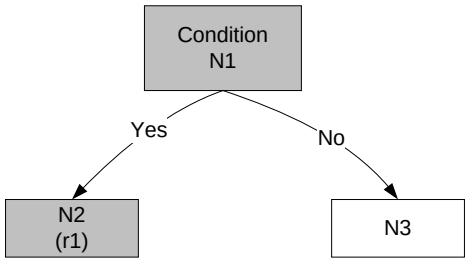

Figure 15. Ownerize a condition node by its successor node of true output link.

IV.5.4 Ownerizing Unownerized Coordination Nodes

As described in Sec. IV.4.3.7, when translating a JOINT DO construct to a responsibility representation, a coordination node is generated for each branch to represent coordination between the branch and other branches. For example, suppose there is a JOINT DO construct as the following:

```
(JOINTDO XOR
   (PAR
      (DO r1 (op1))
      (DO r2 (op2))
   )
   (DO r2 (op3))
)
```

There are two branches in the JOINT DO construct, `(PAR (DO r1 (op1))(DO r2 (op2)))` and `(DO r2 (op3))`. The JOINT DO construct is translated into a responsibility representation as shown in Figure 16. In Figure 16., action `(DO r1 (op1))` is translated to node N1; action `(DO r2 (op2))` is translated to node N2; and action `(DO r3 (op3))` is translated to node N3. Nodes N1, N2 and N3 are ownerized to roles r1, r2 and r3 respectively. To coordinate the execution of the two branches in the construct, two coordination nodes C1 and C2 are generated for the two branches respectively. As shown in Figure 16, C1 and C2 coordinate with each other.

Moreover, whether C1's successor nodes, connected by dependency links L3 and L4, i.e., N1 and N2, are executed depends on the result of C1 coordination with C2; and whether C2's successor node, connected by dependency link L5, i.e., N3, is executed depends on the result of C2 coordination with C1. However, both C1 and C2 are not ownerized to any individual.

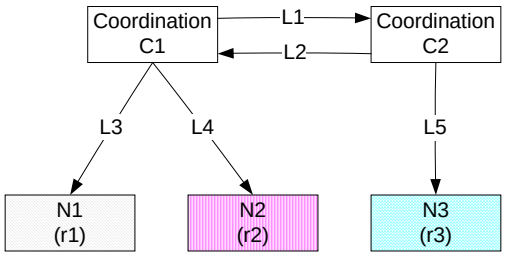

Figure 16. The responsibility representation translated from a joint do construct.

There are two choices to ownerize a coordination node: 1) ownerize it to the owners of all its successor nodes which connected to the coordination node; or 2) ownerize it to the owner of one of its successor nodes which connected to the coordination node.

If we apply the first choice to C1, both r1 and r2 have a copy of C1. To make decision on whether N1 and N2 are executed, these two copies of C1 need to coordinate with C2 so that either N1 and N2 or N3 can be executed. In the meantime, these two copies need to coordinate with each other so that they can make a consistent decision, i.e., both N1 and N2 are executed or none of them.

If we apply the second choice to C1, C1 can be ownerized to either r1 or r2 (suppose r1). r1 executes C1 to coordinate with C2 and the notifies r2 of the result of the coordination. In this way, r1 and r2 have consistent decision on whether N1 and N2 are executed. In essence, that C1 is ownerized to r1 just means that r1 is selected as a coordinator to execute C1 and then notify the result of executing C to others.

From the perspective of communication, the second choice is more efficient. Therefore, we apply this choice to ownerize an unownerized coordination node. That is, an unownerized coordination node is ownerized to the owner of one of successor nodes connected to the coordination node by dependency links. We assume that every

individual is able to notify other individuals of the result of coordination. In reality, some individuals may not be able to notify others or do not have have resource to notify others. If so, the unownerized coordination node could be ownerized to the one that is able and has resource to notify others. More complex handing of these problems is left to future work. For example, Figure 17 shows that the coordination nodes C1 is ownerized to the owner (i.e., r1) of C1's successor node N1 and that C2 is ownerized to the owner (i.e., r3) of C2's successor node N3.

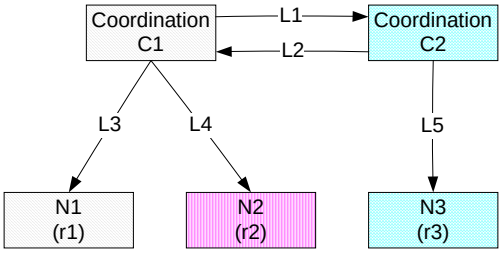

Figure 17. Ownerize coordination nodes in the responsibility representation of Figure 16.

IV.5.5 Algorithm of Decomposing Team Responsibility Graph to Individual Responsibility Graphs

Based on the ideas of ownerizing responsibility components to individuals described in previous sections, an algorithm has been developed to decompose a team responsibility graph to individual responsibility graphs. Suppose that *Resp* is the team responsibility graph of *P* with entrance node, *Start*, and exit node, *End*. The algorithm visits the graph of team responsibility using Depth-First Search from two directions, following the directions of links and starting from the node *Start*, and following the reverse direction of links and starting fromthe node *End*. An array, *VisitedList*, is used to keep track of visited nodes. The algorithm functions as follows:

Algorithm 11. Decomposing a team responsibility to individual responsibilities

DecomposeTeamResponsibility (*Resp, Start, End*)

Begin

Create array *VisitedList*;

```
        ForwardOwnerize(Resp, Start);
        Reset array VisitedList to empty;
        BackwardOwnerize(Resp, End);
        OwnerizeLinks(Resp);
    End

ForwardOwnerize (Resp, Node)
    Begin
        If Node is not in VisitedList
            Add Node into VisitedList;
            For each each output link of Node, Link do
                Let Snode be the destination of Link;
                If Snode is not in VisitedList
                    ForwardOwnerize (Resp, Snode);
                Endif;
            Endfor;
            If Node is not ownerized
                If Node is a condition node
                    Select an ownerized successor node of true output links, Onode;
                    If Onode is not found
                        Select an ownerized successor node of false output links, Onode;
                    Endif;
                Else
                    Select an ownerized successor node of dependency output links, Onode;
                Endif;
                Set the owner of Node as the owner of Onode;
            Endif;
        Endif;
    End
```

```
BackwardOwnerize (Resp, Node)
  Begin
    If Node is not in VisitedList
      Add Node into VisitedList;
      For each each input link of Node, Link do
        Let Pnode be the source of Link;
        If Pnode is not in VisitedList
          BackwardOwnerize (Resp, Pnode);
        Endif;
      Endfor;
      If Node is not ownerized
        Select an ownerized successor node of dependency input links, Onode;
        Set the owner of Node as the owner of Onode;
      Endif;
    Endif;
  End

OwnerizeLinks (Resp)
  Begin
    For each Node not in Nodes of Resp do
      For each output link of Node, Link do
        Set the owner of Link as the owner of Node;
      Endfor;
    Endfor;
  End
```

By ownerizing every component in team responsibility to roles and role variables, a team responsibility graph is decomposed to individual responsibility graphs. In Figure 18, the team responsibility graph of plan `scanandcollect` is decomposed into the

individual responsibility graphs of role r1 and r2. The vertical line in Figure 18 is the boundary between them. After decomposing a team responsibility, a link may connect two nodes in different individual responsibilities. If a link stems from a node in one responsibility to another node in another responsibility; it is called inter-link; otherwise (i.e., it stems form a node to another node in a same responsibility) it is called intra-link.

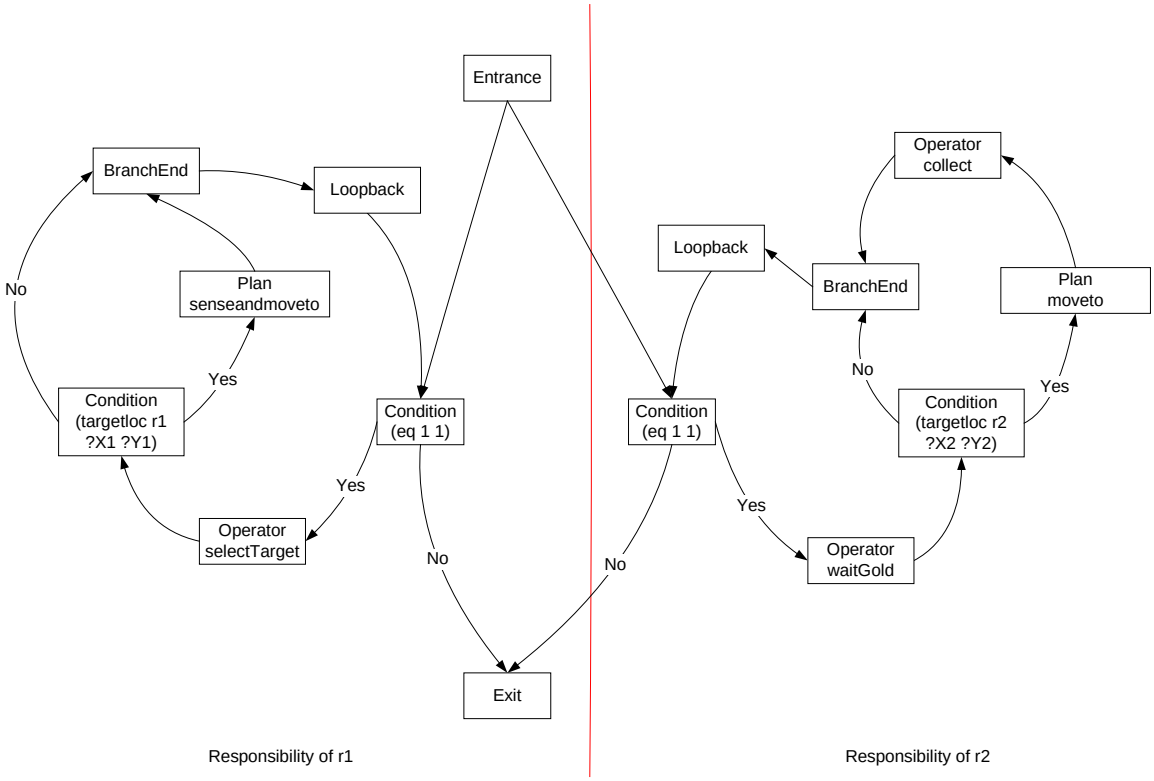

Figure 18. Decomposing the team responsibility of plan `scanandcollect` into individual responsibilities of r1 and r2.

IV.6 Delegating Individual Responsibilities to Agents

Our mechanism of task delegation is realized by delegating individual responsibilities to agents. Our mechanism of task delegation contains two levels of responsibility delegations. The first level is delegating the team responsibilities of all the roles in a role-based plan to the agents invoking the role-based plan before the agents start executing the role-based plan. As described in Sec. IV.4.1, to execute a role-based

plan, a team of agents needs to form a joint intention (i.e., **JIntend***(T, VT, TA, P)*) to perform the actions in the team responsibility of the virtual team in the role-based plan. To form the joint intention, the agents need to decide the team assignment *TA*. Once the joint intention is formed, it drives every agent to intend to perform the actions in the responsibility of the role delegated to the agent (i.e., **Intend***(ag, r, P)*) according to the team assignment *TA*. The second level is dynamically delegating the responsibilities of role variables during the execution of the role-based plan. If a role is selected to fill a role variable, the responsibility of the role variable is dynamically included into the responsibility of the role. Given that a certain agent has intended to the responsibility of the role in the first level, the responsibility of the role variable is implicitly and dynamically delegated to the agent. Therefore, key issues in our mechanism of task delegation are the determining a team assignment for invoking a role-based plan and selecting a role for a role variable.

In this section, we will present our mechanism of determining a team assignment for invoking a role-based plan and our mechanism of selecting a role for a role variable. We will define a notion of role-agent assignment and formalize the admissibility of role-agent assignment to characterize feasible role-agent assignment. Also, we will define a notion of team assignment and formalize the admissibility of team assignment to characterize feasible team assignment. We transform the admissibility of team assignment to a constraint satisfaction problem (CSP) and then present a CSP algorithm to search for admissible team assignment. At last, we will present an algorithm to select a role for a role variable.

IV.6.1 Role-Agent Assignment and Admissibility

We define a role-agent assignment to be a delegation of a role *r* to an agent *ag*, denoted by *Assign*(*r*, *ag*). Once a role *r* is delegated to an agent *ag*, agent *ag* commits to the responsibility of role *r*. Consequently, agent *ag* intends to perform the actions in the responsibility of role *r*, i.e., **Intend***(ag, r, P)* as described in Sec. IV.4.1.

To actually execute the actions in the responsibility of role *r*, agent *ag* must be capable of the actions. Otherwise, agent *ag* is not "qualified" for role *r*, or *Assign*(*r*, *ag*)

is not feasible. We use a notion of admissibility of role-agent assignment to characterize feasible role-agent assignments. A role-agent assignment, *Assign* (*r*, *ag*), is admissible iff *oprequirement*(*r*) $\subseteq$ *capability*(*ag*), where *oprequirement*(*r*) is the qualification of the position *rc* of role *r*, and *capability*(*ag*) is the capability of agent *ag*, i.e., the set of primitive operations (i.e., operators) of which agent *ag* is capable.

We note that an action in the responsibility of role *r* could be a complex action and the capability of agent *ag* is defined in terms of a set of operators. As described in Sec. IV.2.5, the complex action can be rested in individual operations. If agent ag is capable of the individual operations, the agent also can perform the complex action. Therefore, in the definition of admissible role-agent assignment, using primitive operations does not contradict with the fact that the responsibility may contain complex actions.

One may argue that a role in a plan may not perform all operations in the qualification of its position. The reason why we require the agent capable of all operations in the qualification of the position is that, we want to separate role-agent assignment from the detail of the process of a plan. This is consistent with the way that people employ in reality. For example, to fill a job position, a manager makes offer decision based on its qualification and applicant's skills. The manager does not need to consider the details about what is going to be done by this position. For most tasks, the employee for this position only use part of skills required by the qualification of this position. In the other hand, one can define another position that only requires the operations in the plan and define the role as an entity of this position.

IV.6.2 Team Assignment and Admissibility

Suppose that a team of agents *AT* wants to invoke a role-based plan *P* with virtual team (*VT*). We define a team assignment for the agents *AT* invoking the role-based plan *P* to be the aggregation of role-agent assignments of all roles in *VT* to the agents in *AT*, denoted by *Assign(VT, AT)*. It could happen that more than a role is delegated to a single agent. Once the roles in *VT* are delegated to the agents in *AT*, the agents, the agents jointly commit to the team responsibility of *VT*. Consequently, the agents jointly

intend to perform the actions in the team responsibilities of *VT*, i.e., **JIntend***(AT*, *VT*, **Assign**(*VT*, *AT*), *P*) as described in Sec. IV.4.1. To perform the actions in the team responsibility of *VT*, every agent *ag* in *T* intends to perform the actions in the responsibility of role *r* delegated to the agent *ag* according to **Assign***(VT, AT)*, i.e., **Intend***(ag, r, P)*.

To actually delegate the individual responsibilities of the roles in *VT* to the agents in *T*, a few issues need to be considered when making a team assignment. First, according to the definition of team assignment, the team assignment contains a set of role-agent assignments. To ensure that the individual responsibility of each role can be executed by the agent, to which the role is delegated, all of the role-agent assignments in the team assignment should be admissible. Second, as described in Sec. IV.3.1.3, the role-based plan *P* also poses a set of constraints on delegating roles to agents. The team assignment should also satisfy the constraints. Third, the role-based plan *P* may be hierarchical. Suppose a sub-team VT_s of roles may invoke another role-based plan *P′* which contains a virtual team *VT′*. Then, the agents *T′*, to which the roles in VT_s are delegated, jointly commit to the team responsibility of *VT′*. A feasible team assignment is also required to actually delegate the individual responsibilities of the roles in *VT′* to the agents in *T′*.

Similarly, we use a notion of admissibility of team assignment to characterize feasible team assignments. *Assign(VT, AT)* is admissible iff *Assign(VT, AT)* satisfies the following conditions:

1. for every role-agent assignment, *Assign*(*r*, *ag*), in *Assign(VT, AT)*, *Assign*(*r*, *ag*), is admissible;
2. every constraint in the constraint set of plan *P*, *con*, is satisfied under team assignment *Assign(VT, AT)*;
3. for every sub-plan invocation, (**DO** *sub-team P′*), there is an admissible team assignment .

In condition 2, the roles in the virtual team of plan *P* may appear in constraint *con*. To evaluate whether or not *con* is satisfied under team assignment *Assign(VT, AT)*, the

roles in the constraints should be substituted by the agents assigned to the roles according to *Assign(VT, AT)*.

We note, the above formalism is only to search for an admissible team assignment for *AT* invoking *P*, and it occurs when the execution reach the point when *AT* invokes *P*. The statements inside *P*, including condition evaluations and sub-plan invocatioins, have not been reached yet because *P* is not executed yet and *P* is executed only after the admissible team assignment for AT invoking *P*. Checking condition 3 is to ensure that with the found admissible team assignment for *AT* invoking *P*, there will be admissible team assignments for sub-plan invoations of *P* during the future execution of *P* (the team assignment searching for sub-plan invocations occurs when the execution reach them). If condition (3) is not considered when searching for an admissible team assignement for *AT* invoking *P*, it is possible that there will not be admissible team assignements for sub-plan invocations after *P* is executed and when the sub-plan invocations are reached. If so, *P* will fail. Including condition (3) avoids such failures in advance although the execution is at the point to execute *P*. In condition (3), *sub-team* is composed of a subset of roles in the virtual team of plan *P*. The agents that actually invoke plan *P′* are the agents assigned to the roles in *sub-team*. To verify if there is an admissible team assignment for the sub-plan invocation, the roles in *sub-team* need to be substituted for the corresponding agents according to *Assign(VT, AT)*.

It is possible that a sub-plan invocation in *P* is contingent upon evaluation of a condition φ. The contigent sub-plan invocation will be reached only if the future execution of *P* will evaluate φ to true (or false). As the execution is at the point to execute *P*, every statement inside P is not executed yet, so there is no way to determine the evaluation of condition φ. To ensure that with the found admissible team assignment for *AT* invoking *P*, the future execution will not fail to execute the sub-plan invocation, we take a conservative view: we assume that the team must be able to execute all contigent sub-plan invocations. So, we check condition 3 for each sub-plan invocation in *P* no matter whether it is contingent. In this way, we actually check more constraints than necessary. However, if an admissible team assignment satisfies such conservative

constraint, it will ensure that there is an admissible team assignment for each sub-plan invocation when the future execution of *P* reaches the sub-plan invocation.

We note that, in the role-based plan *P*, a subset of roles may form a sub-team to achieve a goal without giving a specific plan (e.g., an Achieve statement or an ACHIEVE alternative in a precondition). The agents to which the roles are delegated to decide what role-based plan to achieve the goal during the execution of the plan P. It is impossible to determine the admissibility of team assignment for that plan before having it. So, that plan is not considered in our formalism of admissible team assignment (for invoking the role-based plan *P*). In our team architecture, we assume that the plan that is dynamically figured out by the agents allows the agents to not only achieve the goal but also find out an admissible team assignment so as to invoke the plan.

One may question why not just decide which role is delegated to which agent during the execution of a role-based plan (i.e., at the first moment when a role invokes an action). The reason why we search for an admissible team assignment before actually executing a role-plan is that, we want to not only decide which role is delegated to which agent, but also have agents form a shared mental state (i.e., **JIntend***(T, VT, TA, P)*) to enforce the execution of the role-based plan to be a team effort; moreover, the shared mental state **JIntend***(T, VT, TA, P)* ensures that every action in the role-based plan will be executed[10] in accordance with the temporal ordering on them unless the termination conditions of the role-based plan are satisfied.

IV.6.3 CSP Algorithm of Searching for Admissible Team Assignments

Suppose a team of agents, $AT = \{ag_1, ag_2, \ldots, ag_n\}$, wants to invoke a role-based plan, *P*. The virtual team of plan *P* is a set of roles, $VT = \{r_1, r_2, \ldots, r_m\}$; and the constraints of plan *P* are a set of conditions, $Con = \{c_1, c_2, \ldots, c_u\}$. In the process of plan *P*, there is a set of sub-plan invocations, $Inv = \{inv_1, inv_2, \ldots, inv_v\}$. inv_i appears as (*DO* vt_i P_i), where vt_i is a subset of roles in *VT* and P_i is a role-based plan. The problem of

[10] A conditional action is executed only when its condition is true.

searching for an admissible team assignment for *AT* invoking plan *P* can be formalized as a constraint satisfaction problem (CSP) [86]:

- the set of variables of the CSP is the virtual team of plan *P*, *VT*;
- the nonempty domain of the CSP is the agents, *AT*. The possible values for each variable (role) are the agents in *AT*;
- the set of constraints of the CSP includes three types of constraints:
- qualification for delegating roles to agents, i.e. for any role r_i, $oprequirement(r_i) \subseteq Capability(Agent(r_i))$, where $Agent(r_j)$ is the agent assigned to r_j;
- the constraints in *Con*;
- an admissible team assignment for each sub-plan invocation in *Inv*.

Various CSP algorithms, including backtracking search, local search [86], and distributed search [130], have been developed to resolve constraint satisfaction problems. Backtracking search algorithms function as a depth-first search to choose values for a variable at a time and backtracks when a variable has no legal value left. Various heuristics can be incorporated into backtracking search algorithms to speed up finding a legal assignment, including forward checking, chronological backtracking, and conflict-directed backjumping. Local search algorithms choose a new value for a variable based on the most obvious heuristic, the min-conflicts heuristic, resulting in the minimum number of conflicts with other variables. Distributed CSP algorithms deploy variables in the multiple platforms. Each platform asynchronously chooses a value for its variable and evaluates relevant constraints. In this way, distributed CSP algorithms speed up the search of a legal assignment.

According to the above formalism, an algorithm of backtracking search has been developed to search for an admissible team assignment for agents invoking a role-based plan. Various heuristics mentioned early may be applied to improve the performance of our algorithm. However, we mainly focus on how to apply CSP algorithms to search for an admissible team assignment. The following is our algorithm of backtracking search for an admissible team assignment:

Algorithm 12. Backtracking search for an admissible team assignment

```
SearchTeamAssignment (AT, P)
   Begin
      Create a set Assignment;
      Return BackTracking_Search(AT, P, Assignment);
   End

BackTracking_Search(AT, P, Assignment)
   Begin
      If Assignment is complete, then
         Return Assignment;
      Endif;
      Randomly select an unassigned role r from the virtual team of plan P, VT;
      For each agent ag in AT do
         If Consistent(P, Assignment, r, ag), then
            Let  Assignment be Assignment + {r = ag};
            Let  result be BackTracking_Search(AT, P, Assignment);
            If result ≠ failure, then
               Return result;
            Endif;
            Let  Assignment be Assignment - {r = ag};
         Endif ;
      Endfor;
      return failure;
   End

Consistent(P, Assignment, r, ag)
   Begin
```

If *ag*'s capability does not contain all operators required by the position of role *r*, then
Return *false*;
Endif;

For each constraint *c* in the constraint set of *P*, *Con*, do
If *Assignment* + {*r* = *ag*} contains all relevant[11] roles in *c*, then
Let *c′* be *c* with all relevant roles substituted by agents according to *Assignment* + {*r* = *ag*};
If *c′* is *false*, then
Return *false*;
Endif;
Endif ;
Endfor;

For each sub-plan invocation $inv_i = (DO\ vt_i\ P_i)$ in *P* do
If *Assignment* + {*r* = *ag*} contains all roles in vt_i, then
Let AT_i be the set of agents that substitutes all roles in vt_i according to *Assignment* + {*r* = *ag*};
Let *SubAssignment* be SearchTeamAssignment (AT_i, P_i);
If *SubAssignment* = *failure*, then
Return *false*;
Endif;
Endif;
Endfor;

Return *true*;
End

[11] If the name of a role is used in a constraint, the role is said to be relevant to the constraint.

We note that the above algorithm is a centralized algorithm. To search an admissible team assignment, one agent serves as a coordinator to run the algorithm, and the coordinator has to know all other involved agents' capabilities. Later in Chapter V, we will describe Role-Based Shared Mental Models. In the Role-Based Shared Mental Model of an agent, the knowledge about other agents' capabilities is maintained. It is possible that the coordinator may not have enough information to evaluate all the constraints. If so, the coordinator queries unknown information from other agents. Later in Chapter V, we will describe a mechanism of Role-Based Proactive Information Exchange. Through this mechanism, agents can proactively exchange information.

The problem of searching admissible team assignment may be resolved by some distributed CSP algorithms. However, the formalism of this problem needs to be modified properly so that the variables are deployed into different platforms. It can be formalized as the agents in *AT* as variables and the power set of *VT* as the domain. However, additional constraints are introduced to make sure all roles in *VT* are assigned to agents and each of them to only one agent. These additional constraints not only increase the complexity of constraint evaluation but also make variables tightly dependent on each other. Consequently, the advantage of asynchronous change in distributed CSP algorithms is lost and a great amount of communication is required to pass new change to other. For these reasons, our teamwork architecture applies the above algorithm by selecting one of the agents invoking a role-based plan as the coordinator and having it execute the algorithm.

IV.6.4 Role Selection

A role selection is the select a role *r* to fill a role variable *?rv*, denoted by **Select**(*?rv, r*). In Sec. IV.2.4, a role variable is defined as an entity which is selected from a set of entities and associated with a bag of temporally ordered actions. Role variables are mainly used to represent dynamic action association. If a role *r* is selected to fill a role variable *?rv*, the actions associated with the role variable *?rv* are dynamically associated with role *r*. Moreover, the responsibility of role variable *?rv* is dynamically included in the responsibility of role *r* as described in Sec. IV.4.1. The agent to which role *r* is

delegated intends to execute the actions in the responsibility of role variable *?rv*, i.e., **Intend***(ag, ?rv, P)*, as described in Sec. IV.4.1.

In Sec. IV.3.2.3, we have given the syntax of role variable declaration, which specifies the information about a role selection for a role variable, including the selection scope (i.e., a list of roles) and the selection constraints. Moreover, as we defined in Sec. IV.2.4, the capability requirement of every role in the selection scope must include all actions (primitive operations) associated with the role variable. Suppose the capability requirement of a role in the selection scope does not include all actions associated with the role variable. If the role is selected to fill the role variable, then the agent to which the role delegated may not be capable to do some action(s) associated with the role variable. To prevent this oddity, a plan can be statically checked if all role variables in the plan are well defined, i.e., for every role variable, all roles in the selection scope can perform the actions associated the role variable. Further, if all role variables are well defined, then the agents to which the selected roles are delegated must be able to execute the actions associated with the role variables. For this reason, we only consider the selection constraints but not the capability issue when resolving role selections.

Suppose there is a role variable *?rv* in a role–based plan *P*. The selection scope is a set of roles *RS*. The selection constraint is *Cons*. A team of agents *T* executes the role-based plan *P* with a team assignment *TA*. The following algorithm is used to decide which role would be selected to fill role variable *?rv*.

Algorithm 13. Selecting a role for a role variable

RoleSelection (*?rv*, *RS*, *Cons*, *TA*)

```
Begin
    Loop
        If all roles in RS have been tested, then
            Return failure;
        EndIf;
        Randomly select an untested role r from RS;
        Let ag be the agent to which role r is delegated according to TA;
```

If all constraints in *Cons* are satisfied with *?rv* substituted by *ag*, then

Return *r*;

Endif;

Mark role *r* as a tested role;

EndLoop;

End

We note that the admissibility of team assignment did not include checking the constraints of role selections. As we have explained in Sec. IV.6.2, a main purpose of searching for an admissible team assignment before actually executing a role-based plan is to allow agents to form a shared mental state **JIntend***(T, VT, TA, P)*) to enforce the execution of the role-based plan to be a team effort. There are two reasons why checking the constraints of role selections was not included in the admissibility of team assignment. First, it may be impractical to include checking the constraints of role selections in the admissibility of team assignment. While a role is used to express static action associations, a role variable is an intermediate to express dynamic action associations. The decision on which role is select to execute the actions associated with the role variable depends on concrete situations. This means, the evaluation of the constraints on a role selection for a role variable may dynamically change with the execution of a role-based plan. Given the evaluation of the constraints are dynamic with the plan execution, the evaluation cannot be accomplished before the plan execution.

Second, even though the admissibility of team assignment does not include checking the constraints of role selections, agents, through searching for an admissible team assignment, still can from a shared mental state **JIntend***(T, VT, TA, P)*) to enforce the execution of the role-based plan to be a team effort. As we have explained early in Sec. IV.2.4, a role variable is an intermediate to express dynamic action associations and a role must be dynamically selected to be associated with the actions which are associated with the role variable. So a role variable is only a reference to some role, but the decision about to which role the role variable refers is dynamically made. Also, as described in Sec. IV.4.1, the responsibility of a role variable will be included in the responsibility of

the role selected to fill the role variable. Once a team of agents has formed the shared mental state **JIntend***(T, VT, TA, P)*) based on an admissible team assignment *TA*, the joint intention implicitly contains an intention to perform the actions in the responsibilities of all role variables in plan *P*.

Ones may argue that assuming that a role must be selected to fill a role variable is a little restrictive, and that it is possible that no role can be selected to fill a role variable in some situations. Not to say that this is contradictory with our philosophy that a role variable is only a reference to some role for dynamic action associations for a moment, we can take a remedy in our formalisms of responsibility. That is, the joint intention to execute the actions in a team responsibility can be formed relative to a condition as well as the termination conditions of plan *P*. If no role is selected for a role variable, the condition turns to be false. Once the condition becomes false, the joint intention is dismissed.

IV.7 Reasoning on Reuse of Role-Based Plans

Role-based plan can be reused in two ways. First, the users of RoB-MALLET use existing role-based plans to code new role-based plans for different goals. Similar to procedures or methods in other programming language, existing role-based plans can be reused to specify new role-based plans (described in previous sections). Second, planning algorithms [86] can use role-based plans as complex actions, as well as operators that can be performed by individual agents, to form a course of actions to achieve certain goals.

While our research goals do not include developing new planning algorithms, role-based plans are amenable for use (with slight modifications which we will describe) by a number of existing planning algorithms. In this section, we explained how our role-based plans could be reused by planning algorithms (forward state-space search and backward state-space search). In addition, we have implemented a simple plan search algorithm that allows us to demonstrate the capabilities of re-use of role-based plans.

IV.7.1 Reusing Role-Based Plans in Planning Algorithms

Planning is the process of coming up with a course of actions that will achieve a certain goal, based on states, goals, and actions with preconditions and effects [86]. Planning algorithms decompose a problem into a sequence of sub-problems, typically a sequence of states, and then find solutions these sub-problems. For example, STRIPS [61] decomposed a problem into a sequence of space states based on the difference caused by relevant operators. Forward state-space search starts from the initial state and considers possible sequences of actions until it finds a sequence that can reach a goal state. In contrast, backward state-space search starts from the goal state and considers possible regression states of relevant actions (an action is relevant to a goal if it achieves the goal) until it finds a sequence of relevant actions that regresses the goal to the initial state. Once a course of actions is found, the agent executes them to achieve its goal.

In our teamwork architecture, both operators and role-based plans are actions that change the world model from the states specified by preconditions to the states specified by effects. From this perspective, both of them can be used as actions for planning purposes.

An operator is an atomic action and only needs an agent or agents to execute it. Suppose an agent has found a course of operators for a goal. If an agent is capable of performing all operators, the agent is able to use the course to achieve its goal. Moreover, plan algorithms search a course of operators for a goal on the basis of operators that the agent is capable of performing.

However, a role-based plan is executed by multiple agents. To be a proper action, it is necessary for a role-based plan to transform the world from current state to a goal, i.e., its precondition is satisfied under current state; and its effect implies the goal. Moreover, to jointly execute the role-based plan, the agents need to form a joint intention to execute the actions in the team responsibility corresponding to the role-based plan. To form the joint intention, an admissible team assignment is needed so that the individual responsibilities can be delegated to the agents to actually execute the role-based plan.

Based the above analyses, role-based plans can be applied as actions for planning algorithms as follows:

- For forward state-space search, the role-based plans are applicable to a state are all those whose preconditions are satisfied and that allow the agents to have admissible team assignments to invoke the plans;
- For backward state-space search, the role-based plans are applicable (i.e., relevant) to be a state are those whose effects are satisfied under current regression state and that allow the agents to have admissible team assignments to invoke the plans

IV.7.2 Algorithm of Achieving a Goal by a Role-Based Plan

In this section, we give a simple version of planning algorithm which only searches for a role-based plan from a library of role-based plans to achieve a goal. It neither searches operators nor conduct state-space search. However, it can be applied to forward and backward state-space search to check which role-based plan(s) is (are) applicable to a state.

According to RoB-MALLET syntax, the following information is required to invoke a role-based plan:

- plan name,
- arguments for the plan,
- and the agents that invoke the plan.

If a team of agents wants to achieve a goal by a role-based plan, the agents need to find the plan name of a proper plan and the arguments for invoking the plan. The proper plan can be found by searching the plan library and checking if the effects of a plan imply the goal, if its preconditions are currently satisfied, and if there is an admissible team assignment for the agents to invoke it.

The effects of a role-based plan depend on the variables in the effects. Different arguments (i.e., the values of variables) of invoking a plan can result in different effects. If the effect of a role-based plan implies the goal, the arguments can be found by

unifying its effect and the goal. For example, suppose agents has a goal (pred 5 6) and there is a plan P1 which has variables ?x and ?y and an effect (pred ?x ?y). By unifying the effect (pred ?x ?y) and the goal (pred 5 6), we can know the effect of plan P1 may imply the goal, and P1 needs to be called with arguments 5 (for ?x) and 6 (for ?y) to achieve the goal.

The preconditions of a role-based plan also depend on the variables in the preconditions. Different arguments of invoking a plan can lead to different evaluation of its preconditions. To invoke the role-based plan, the arguments must satisfy the preconditions.

Thus, the arguments can be determined along with checking whether the preconditions are true and whether the effects imply the goal. To check whether the effects imply the goal, unification between the effects and the goal can be made. If some substitutions are found, the goal is implied by the effects. Moreover, some arguments are found if their corresponding variables are included in the substitutions. To check if the preconditions are satisfied, unification between the preconditions and agent's beliefs can be made. If some substitutions are found, the preconditions are satisfied. Similarly, some arguments are found if their corresponding variables are included in the substitutions. It is possible that some arguments are still not found even though the goal is implied by the effects and the preconditions are satisfied. If so, they do not affect the invocation of the plan no matter what they are.

Suppose there is a library of role-based plans. Given a goal and a team of agents, the following algorithm can find a role-based plan to achieve the goal by the agents invoking the plan:

Algorithm 14. Searching a role-based plan for agents to achieve a goal

Achieve (*AT, Goal, PlanLibrary*)

Begin

Create a binding set *Bindings*;

For each plan *P* in plan library *PlanLibrary* do

Reset *Bindings* to empty;

If the effect of *P* implies *Goal*, then
Unify the effect of *P* with *Goal* and add all substitution to *Bindings*;
Substitute all identifiers in the precondition of *P* with their values according to *Bindings*;
If the substituted precondition is satisfied, then
Unify the substituted precondition with the states that satisfies it and add all substitution to *Bindings*;
If SearchTeamAssignment (*AT, P*) ≠ *failure*, then
Let *Args* be the list of values of the variables of *P* according to *Bindings*;
Return *(DO AT (P Args))*;
Endif
Endif;
Endif;
Endfor;
Return *failure*;
End

Later in Sec. V.6.3, this algorithm will be used to find a role-based plan for provide helping behaviors.

IV.8 Discussion and Summary

IV.8.1 Discussion

In RoB-MALLET, a role-based plan is specified in terms of roles and role variables. Agents are explicitly excluded from the specification of role-based plans. Role-based plans can be reused by different agents. In MALLET, a plan is specified in terms of agent variables and agents. Using agents in the specification of plans prevents from the plans from being reused by other agents. Some modification can be made in MALLET to explicitly exclude agents in the specification of plans. In the following discussion, we assume that agents are excluded from the specification.

In MALLET, *Agent-bind* statements are used to specify the constraints on selecting agents for agent variables in the process of a plan. In RoB-MALLET, a constraint set in par with the process of a role-based plan specifies the constraints on delegating roles to agents; and *Select* statements are used to specify the constraints on selecting roles for role variable (eventually agents are selected for the role variable through the roles).

From the perspective of expressivity, RoB-MALLET can express any plan in MALLET by an equivalent role-based plan. Suppose there is a plan in MALLET. For any *Agent-bind* statement with static constraints, it agent variable can be translated to a role, and its constraints can be added to the constraint set of the corresponding role-based plan in corresponding formats. For any *Agent-bind* statement with dynamic constraints, its agent variable can be translated to a role variable, and the *Agent-bind* statement is translated to a *Select* statement.

Moreover, roles and role variables bring up a set of advantages over agent variables. First, RoB-MALLET distinguishes static constraints on selecting agents (i.e., the constraint set) form and dynamic constraints (constraints of selecting roles for role variable) and then solves all static selections together as an admissible team assignment by a CSP algorithm. We note that the selections with dynamic constraints must be solved dynamically and thus they cannot be solved together with others. However, if the static selections are expressed by a set of *Agent-bind* statements, each *Agent-bind* statement must be solved separately and it is a CSP unless additional mechanisms are used to distinguish *Agent-bind* statements with dynamic constraints and merge *Agent-bind* statements with static constraints to a CSP. As we know, backtracking take places when a variable has no proper value. Now each *Agent-bind* is a separate problem, backtracking could no longer take place. This may prevent all selections from be solved.

Second, roles and role variables not only serve as notions to express team activities but also enable mechanisms of task decomposition and task delegation. In MALLET/CAST, when an operator is to be executed, the agent which is selected to fill the agent variable associated with operator are delegated to execute the operator. As we have shown in previous sections, a plan is more than an aggregation of operators. There

are many tasks other than operators that need to be executed, for example, condition evaluation and the flow control to ensure the temporal orders on operators. Moreover, through our task decomposition and task delegation, agents execute a role-based plan by a joint intention which ensures the execution of the role-based plan be a team effort.

Third, role and role variables enable a mechanism of using role-based plans together with operators by planning algorithm. As part of our task decomposition mechanism, we have formalized a notion of admissibility of team assignment. When a team of agents invokes a role-based plan, the agents need to have an admissible team assignment so as to form a joint intention to execute the plan. In planning algorithms, whether a role-based plan can be applicable to a state depends on whether the agents have an admissible team assignment as well as whether the preconditions or the effects of the plan match the state.

IV.8.2 Summary

The concepts of position, role and role variable have been formalized to capture the behavioral aspects of roles and to enable computational mechanisms in teamwork. A position characterizes the behaviors of a class of entities. A role is an entity of a position with associated actions and temporal orders on them. A role variable represents dynamic action associations with one of a set of roles. A role-based plan specifies team activities in terms of roles and role variables. The process of a role-based plan is composed of temporally ordered actions associated with roles and role variables. By using conceptual roles and role variables, instead of concrete agents, role-based plans can be reused for different situations. The RoB-MALLET syntax allows specifying positions, roles, role variables, and role based plans. In the processes of a role-based plan, the RoB-MALLET syntax allows expressing actions associations with roles and role variables and temporal ordering on the actions.

Task decomposition and task delegation can be enabled through a notion of responsibility. The notion of responsibility captures the temporally ordered actions to be executed and their impacts on agents' mental states. A role-based plan can be translated

into a team responsibility graph and then the team responsibility graph can be decomposed to individual responsibility graphs of roles and role variables. Once a team of agents invokes a role-based plan, the agents jointly intend to execute the actions in the team responsibility. The execution of the actions in the team plan is realized by delegating roles to agents. A notion of admissibility of team assignment characterizes feasible delegations of all roles in a role-based to the agents invoking the role-based plan. Once an admissible team assignment is found, the agents intend to execute the actions in the responsibility of the roles delegated to the agents. During the execution of a role-based plan, an agent intends to execute the actions in the responsibility of a role variable once the role delegated to the agent is selected to fill the role variable.

Based on the notion of admissibility of team assignment, planning algorithms can use role-based plans together with operators to search for a course of actions for achieving a certain goal. To apply a role-based plan to a state, agents must have an admissible team assignment for the role-based plan so as to form a joint intention to actually execute the role-based plan.

CHAPTER V
ROLE-BASED SHARED MENTAL MODELS

The notion of mental model has been used as an explanatory mechanism in various disciplines over decades [12, 65, 85, 87]. It is very broad and includes many aspects, including how to represent the environment as knowledge, how to interpret knowledge, and how to reason. In this chapter, we introduce our concept of role-based shared mental model, relate it to previous models in the literature, elaborate how we construct and maintain our models from the concepts introduced in Chapter IV, show how it can be used to maintain mutual awareness, and how it can be used to support various forms of useful reasoning.

V.1 Introduction

V.1.1 Relationship of Role-Based Shared Mental Models to Previous Models

The ability to reason includes the evaluation of assumptions and hypotheses, making decisions about courses of actions, the pursuit of arguments and negotiations, finding the solutions of problems and many other aspects. Johnson-Laird [54] summarized mental models as working models in human minds by which human beings understand the world and interact with it. Following the doctrine of functionalism of mental models [54], a mental model must be constructive, including how does it represent the environment (what processes construct and interpret it? what concepts does it contains and what are its basic structures?) Moreover, a mental model and the machinery for constructing and interpreting it must be computable. Once a mental model is constructed, it can act as a framework to describe, explain, and predict the events in the environment.

In recent years, the notion of shared mental model extended mental model to a context of a team. Cannon-Bowers et al. suggested [12] that shared mental models are "knowledge structure held by members of a team that enable them to form accurate explanations and expectations for the task, and in turn, to coordinate their actions and

adapt their behavior to demands of the task and other team members". Klimoski and Mohammed [55] insisted that "there can be (and probably would be) multiple mental models co-existing among team members at a given point in time". Stout et al. [97] exposed that effective planning increased the shared mental models among team members, allowed them to utilize efficient communication strategies during high-workload conditions, and resulted in improved coordinated team performance. Cannon-Bowers and Salas [11] clarified four critical aspects of "shared cognition":

- first, what must be shared is task-specific knowledge, task-related knowledge, knowledge of teammates and attitudes/beliefs;
- second, "shared" can mean overlapping (refers to situations where two or more team members need to have some common knowledge, e.g., a surgeon and a nurse do not have identical knowledge, but portions of their knowledge need to be common to them), similar/identical (refers to situations where team members need to hold similar/identical knowledge, e.g., team members hold mutual beliefs), complimentary (refers to situation where team members holds dissimilar knowledge but the dissimilar knowledge draw similar expectations for performance, e.g., agents with different capabilities bring their specialized expertise to their tasks), or distributed (refers to situations where knowledge is effectively apportioned across team members, e.g., in military battlefields, tasks are so complex that it is impossible for any single team member to hold all knowledge and the knowledge must be distributed across team members);
- third, "shared cognition" can be measured by assessing structure and/or content of team member knowledge;
- and fourth, "shared cognition" can lead to better team processes and better task performance.

In role-based shared mental models, we adopt the overlapping, complementary and distributed view of "shared." That is our agents will be distributed, but have some knowledge that is individual and complementary to the team and some that is shared.

Smith-Jentsch et al. [93] and Langan-Fox et al. [58] suggested that one develop mathematical mechanisms of similarity of shared mental models and further explore the relationship between similarity and team performance. Both have also conducted several experiments and indicated that higher similarity among shared mental models leads to better team performance. Nevertheless, Langan-Fox et al. [57] also pointed out that formalizing the similarity of shared mental models, measuring the relationship between teamwork effectiveness and the similarity of shared mental models, and developing efficient representations of shared mental models (which support effective teamwork with low similarity) remain challenging future research directions on shared mental models. An interesting study by Levesque et al. [60] reported a contrary result that the similarity of share mental models decrease with the increase of role specialization over the course of software development.

Our teamwork architecture utilizes an efficient representation of shared mental models, Role-Based Shared Mental Models (RoB-SMM), to facilitate effective teamwork among agents. A RoB-SMM consists of a variety of team knowledge, including the knowledge of domain, capability, role-based plans, beliefs and inference rules, goals, team structure, and team process. Each agent will have a RoB-SMM, and the union of these models for a team will be the overlapping, complementary distributed mental model of our role-based plans.

The knowledge of a team process plays a critical role in RoB-SMMs because it enables each team member to be aware of what other team members are doing (perhaps only partially), team members to coordinate to make critical decisions and to resolve conflicts, and various reasoning to improve team effectiveness, such as anticipating information needs of other team members and providing helping behaviors to other team members. Each agent in our teamwork architecture maintains the team process by keeping a copy of team organization and an individual process. A team organization contains the relationship between shared goals and the plans to achieve the goals, and team assignment to execute the plans (i.e., which agents play which roles in the plans). The team organizations in the RoB-SMMs of different agents overlap, but they are not

necessarily identical, depending on the level of mutual awareness the team wants to maintain (see Sec. V.4). The individual processes are complementary and each is represented by an extended Petri net, RoB-CAST-PN (see Sec. V.5.2). Each only contains the actions related to the agent possessing it. The individual processes in the RoB-SMMs of different agents are thus different. Therefore, comparing the shared mental models in [127, 128], which maintain a consistent and detailed copy of the team process in each agent, the knowledge of team process in RoB-SMMs has a low similarity, i.e., a low level of overlapping among team organizations.

The representation of a team process in our RoB-SMM is compatible with the mechanisms of task decomposition and task delegation described in Chapter IV. When a team of agents invokes a role-based plan to achieve a goal, the roles in the plan are delegated to the agents. Depending on the level of mutual awareness maintained, the team organization of an agent records the goals, the plans, and the assignments of all members, or only certain subteams, of the teams. Each involved agent dynamically composes its individual process by translating the responsibility of the role(s) delegated to the agent into an extended Petri-Net representation

Our RoB-SMM simulates effective teamwork even though it only maintains a low level of overlapping. Moreover, it improves the effectiveness of teamwork from two aspects: time performance and communication performance. Experiments in Chapter VI will illustrate the improvement. Our RoB-SMM can facilitate mutual awareness among agents. Various reasoning based on mutual awareness can help achieve effective teamwork. Through mutual awareness agents can reason about the information needs of other team members and proactively exchange information among the agents, and agents can determine helping needs of other team members and proactively help them.

V.1.2 Structure of the Chapter

In Sec V.2, we will elaborate a variety of knowledge kept in a RoB-SMM. In Sec. V.3, we will explain why each agent maintains a team process by a team organization and an individual process. In Sec. V.4, we will describe what a team organization is and what team mental states are maintained. In Sec. V.5, we will discuss the advantages of

Petri nets and analyze the problems in existing CAST-PNs (previous works of our MURI group [127, 128]), which maintain a consistent and detailed team process in every agent. To make use of our mechanisms of task decomposition and delegation described in Chapter IV and tackle the problems existing in CAST-PNs, we will formalize an extension of Petri nets, RoB-CAST-PN, to represent individual processes. We will explain how an individual process is maintained as agents achieve goals and invoke role-based plans, including initializing and finalizing invocations, translating individual responsibilities into RoB-CAST-PNs, and dynamically composing RoB-CAST-PNs. We will depict the interactions between team organizations and individual processes. The algorithm for executing individual processes will be given. In Sec. V.6, we will explain how agents achieve mutual awareness through team organizations. Two reasoning mechanisms based on mutual awareness will be developed to achieve effective teamwork, including role-based proactive information exchange and role-based proactive helping behaviors. The formalisms and algorithms of these two reasoning mechanisms will be elaborated.

V.2 Role-Based Shared Mental Models

In our teamwork architecture, a role-based shared mental model contains critical team mental states, including mutual beliefs, shared goals, team plans and joint intentions. In Chapter III, we have explained how these team mental states are expressed by MALLET constructs. These mental states specified by MALLET statements need to be properly represented in the RoB-SMM. Besides these mental states, MALLET also provides constructs to specify the knowledge of the domain (such as the specification of operators), agents' capabilities, and team structures. Such knowledge is also shared by agents and thus needs to be properly represented in the RoB-SMMs. As described in Chapter IV, we have introduced concepts related to roles into our teamwork architecture; plans are specified in terms of roles. Role-based plans are shared by the agents. Agents invoke role-based plans to achieve their goals. The role-based plans specify a team process. At any point in time, however, an individual agent may be involved in multiple roles in multiple plans. In order to properly represent the RoB-SMM for that agent, the

composite of these roles and plans must be maintained, together with their relationship to this and other agents.

More specifically, a RoB-SMM maintains the following information:

- knowledge of domain application. This includes the operators in the domain and their specifications and the positions in the domain and their specification. Operators and positions can be directly specified by MALLET.
- knowledge of capability. This represents which operators each agent in the team can perform. Capabilities of agents can be directly specified by MALLET;
- knowledge of role-based plans. This includes a set of role-based plans. All role-based plans can be specified by our RoB-MALLET and properly organized in the system (e.g., organized in a directory of file system). These three types of knowledge are static while the rest knowledge evolves dynamically as agents achieve goals and/or execute plans;
- knowledge of beliefs and inference rules. Beliefs and rules can be directly inserted or deleted by ASSERT or RETRACT constructs in our RoB-MALLET. Also, operators and plan may modify beliefs.
- knowledge of goals. This includes the goals shared by (sub)teams of agents. In our RoB-MALLET, the goals can be specified in two ways, ACHIEVE constructs or operator/plan invocations. The effects of an operator/plan are regarded as the goals of the agents invoking it;
- knowledge of team structure. This represents all teams in the system and what agents are included in the teams. Our RoB-MALLET specifies team structure by the TEAM construct. In the process, (sub)teams may be formed by a list of roles to invoke (sub)plans or team operators or to achieve some goals;
- knowledge of team process. This represents the status of the execution of invocations, which helps agents to directly decide their next actions. In our role-based shared mental models, it is represented by the team organization and the individual process. Team organization is the common knowledge of a team of agents. It contains shared goals, the plans to achieve the goals, and team

assignments to execute the plans (i.e., which agents play which roles in the plans). an individual process is the portion of a team process relevant to an agent. It only contains the actions related to the agent. We will give the rationale for such representation in Sec. V.3. More details about team organization and individual process will be given in Sec. V.4 and V.5.

V.3 The Rationale for the Representation of Team Processes in RoB-SMMs

The representation of team processes is one of critical issues which affect the performance of teamwork. The representation of team processes in RoB-SMMs is efficient in supporting effective teamwork. In next paragraphs, we will start analyze the problems with the representation of team processes in previous work, i.e., CAST-PN [127, 128]; we will explain how our RoB-SMMs accommodate the mechanisms of task decomposition and delegation (described in Sec. IV.4 and IV.5) to tackle those problems; and we will discuss what should be included in the representation of team process in RoB-SMMS correspondingly.

In previous work [127, 128] of our MURI group, the knowledge of a team process is represented by a Petri net, called CAST-PN, which is translated from the invoked team plan. As the knowledge of team process is a shared knowledge among agents, each agent has a copy of the CAST-PN. As a team plan is executed, each agent updates its CAST-PN and keeps its CAST-PN consistent (partially in CAST 3.0) with others. Such representation can hook multiple agents to perform teamwork and facilitate proactive information exchange.

However, a few problems were caused by maintaining consistent CAST-PNs. First, the maintenance of consistent CAST-PNs (partially in CAST3) requires broadcasts and thus causes a large amount of unnecessary communication. Second, to prevent and resolve global conflicts, synchronization is required for executing each copy of CAST-PN. As a result, the parallelism of plan execution is limited. Third, it is not flexible

enough to support simultaneous plan execution[12]. The CAST-PN is directly translated from a team plan. If multiple plans are invoked at the same time and an agent may participate in more than one of them, it is hard to use CAST-PN to represent the participation of multiple plans. Also, the proactive information exchange is limited to being within a single plan invocation at the top level. One may argue a team plan can be written to include all simultaneous plans. However, doing so is very inflexible, particularly as the simultaneous plans cannot all be pre-determined in a dynamic domain.

Moreover, CAST-PN requires all agents know all team processes. In many domains, it is impossible for a team member to know all team processes. A team member typically only knows the portion related to itself. A team member may know the overall of other team members' portions, but it does not necessarily know their exact execution status.

The task decomposition and role delegation explained in Chapter IV provide the basis to resolve the above problems. A team process can be decomposed into individual responsibilities of the roles in a role-based plan; and in this case, an agent only needs to know the portion of team process related to the role(s) delegated to it, i.e. the individual responsibility of the role(s). Moreover, if the execution model is constructed based on individual responsibilities, it can avoid the maintenance of the consistency of petri nets and global checking as is done in CAST-PN. In particular the execution model based on individual responsibilities must be compositional because multiple roles in a role-based plan could be delegated to a single agent. Such compositional execution model allows an agent to participate multiple simultaneous plan invocations.

On the other hand, individual responsibilities may contain coordination with others as explained in Chapter IV, such as sub-plan invocation and role variable selection. If the execution model is constructed on the basis of individual responsibilities, an agent needs to know with which agent(s) it coordinates and how the coordination takes place when the execution approaches certain coordination. Thus, the knowledge about the role delegation for the plan invocation should be shared by all involved agents while each

[12] Simultaneous plan execution is the execution of multiple plans, which are not in the hierarchy of a hierarchical plan. We treat the execution of a hierarchical plan as the execution of a plan.

agent constructs its execution model based on individual responsibilities of the roles delegated to it.

Based on the above analyses, the knowledge of team processes can be represented in terms of roles and the delegation of roles to agents. The execution model of team process for each agent can be constructed by the portion related to the agent, i.e., the responsibility of the role(s) delegated to the agent. The execution model related to the agent is called an individual process of the agent in our RoB-SMM. The overlapping of a team process maintains plan invocations, their individual portions in the plan invocations, and the relationships between agents and the individual portions. This overlapping is called team organization in our RoB-SMM. In the next two sections, we will give more detail how the knowledge of a team process is represented by team organizations and individual processes

V.4 Team Organization

A team organization in our RoB-SMM represents the hierarchy of shared goals, the plans to achieve the goals, and the team assignments for executing the plans. The agents may be involved in multiple goal achievements at a time. The goals may consist of a hierarchy of sub-goals. The agents invoke role-based plans to achieve these (sub)goals. To actually execute these role-based plans, the roles in the role-based plans are delegated to the agents. Corresponding to the hierarchy of (sub)goals, a team organization is a tree structure. Each node in the tree structure is a tuple (G, φ, P, S, C), where G is a team of agents, φ is a shared goal of G, P is a role-based plan by which the agents in G achieve φ, S is the team structure of G invoking P, and C is the set of nodes corresponding to the sub-goals of φ. The team structure of G invoking P is the mapping from the individuals (roles and role variables) in the plan to the agents invoking the plan, including the team assignment that G invokes P, $Assign(VT, G)$ (VT is the virtual team in P), and the role selections during the execution of P. Team organization is illustrated by Figure 19.

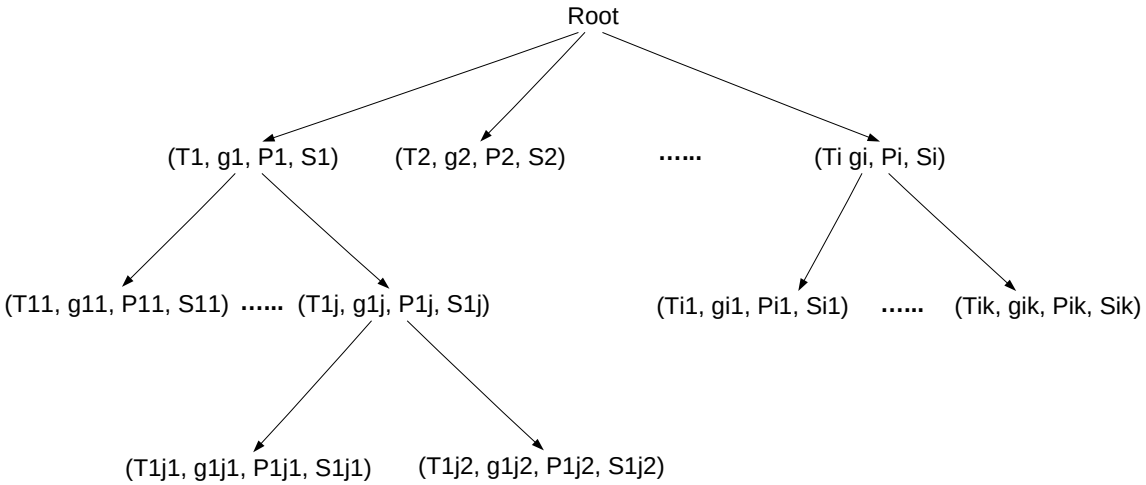

Figure 19. An example of team organization.

A goal in a team organization is a logical predicate or condition. A goal can be stated explicitly by an ACHIEVE construct in the RoB-MALLET or implicitly by the effects of plans that teams are directed to do (i.e., by START or DO constructs in the RoB-MALLET). One might argue that the effects of operators also can specify a goal and a role-based plan may contain operator invocations. However, the goals specified by the effects of operators are not necessary to be recorded team organizations. The reasons are:

- Suppose there is an operator invocation *OInv* in a plan invocation *PInv*; once the team structure for *PInv* is determined, the performer of *OInv* is also determined; further, which agent(s) is(are) to achieve the effect of the operator corresponding to *OInv* is also determined.
- Our teamwork architecture assumes that invocations at the top level are initialized by a role-based plan invocation, starting a plan by a START statement or using a role-based plan to achieve a goal specified by an ACHIEVE statement. Given a role-based plan and the team structure of perform the plan, an agent can derive all sub-goals specified by operator invocations in the plan and which agent(s) shares the goals.

According to such interpretation of goals, the goal corresponding to a hierarchical plan hierarchically contains sub-goals corresponding to its sub-plans.

The agents may be working on multiple goals at the same time. These goals are independent and each corresponds to a node in the team organization. To accommodate such simultaneous goals, a special node is used as the root of the tree structure of team organization which points to simultaneous goals at the top level. In Figure 19, the agents are involved in a set of simultaneous goals $g_1, g_2, \ldots, g_i$.

By the team organization, critical team mental states, such as shared goals and joint intentions, and the relationships between them are maintained in agents' shared mental models. The tuple (G, φ, P, S, C) is used to record two critical team mental states: 1) a shared goal (note: a shared goal in our teamwork architecture is similar to joint persistent goal in [22, 59]), denoted *Goal*(G, φ), and 2) a joint intention, denoted **JIntend***(G, VT, TA, P)* (described in Sec. IV.4.1), where *VT* is the set of roles in P, *TA* is the part of the team assignment in team structure S. To achieve their shared goal φ, the agents in G jointly intend to perform plan P by delegating the roles in P to them according to team assignment *TA*. Moreover, the relationships between goals and their sub-goals are maintained. In Figure 19, goal g_{1j} consists of sub-goals g_{1j1} and g_{1j2}.

Each agent has a copy of the team organization. However it is not necessary that all copies be identical. Each agent may maintain its team organization in one of two different modes, relevant and full tracking. If an agent maintains its copy of team organization in the mode of relevant tracking, it just maintains the goals in which it is involved and the role-based plans and team structures to achieve these goals of performing the plans. If an agent maintains its copy of team organization in the mode of full tracking, it maintains all goals, no matter whether it is involved in them, and the role-based plans and team structures to achieve these goals of performing the plans. Later, we will show that there is a tradeoff between efficiency and the level of reasoning that can be performed based on the tracking mode.

These two modes result in different overlapping on the copies of team organization. With the first mode, any two copies of team organization may be different from each

other, and less synchronization to maintain team is required. With the second mode, all agents have identical copies of team organization, and more synchronization to maintain team is required.

To make the agents accomplish their teamwork, the agents at least maintain the goals which they involve and corresponding role-based plans and team structures. In later sections, more details about individual process and the interaction between team organization and individual process will be given to illustrate how team process in our teamwork architecture directs agents to perform teamwork. The agents in the second mode can monitor what other agents are doing, even if the agents do not participate in them. This allows agents to perform various reasoning to achieve effective teamwork, for example, proactive information exchange across simultaneous plans and proactive helping behavior. We will explain proactive information exchange in Sec. V.6.2 and proactive helping behavior in Sec. V.6.3.

V.5 Individual Process

The individual process of an agent is the portion of a team process related to the agent. It is represented by an extension of Petri nets, called RoB-CAST-PN (Role-Based CAST Petri Net). When an agent, together with other agents, invokes a role-based plan, the agent translates the individual responsibility graph of the role delegated to the agent into a RoB-CAST and dynamically composes its individual process with the RoB-CAST. The agent executes the role-based plan by executing the RoB-CAST-PN in its individual process.

In this section, we first analyze the advantage of Petri nets and the problems with previous representation of team processes (CAST-PN). Based on these analyses, we describe our RoB-CAST-PN which can solve the problems with CAST-PN. Then we explain how an agent maintains its individual process, including initializing and finalizing invocations, translating individual processes into RoB-CAST-PNs, and dynamically composing its individual process. As an agent represents a team process by a team organization and an individual process, we will describe the interactions between

its team organization and individual process. At last, an algorithm will be presented to show how an agent executes its individual process.

V.5.1 Advantages of Petri Nets

In teamwork, team members need to coordinate with each other and make team decisions. Team decision making is complex and effective team decision making demands sophisticated and powerful modeling tools. Because of three attractive properties of Petri nets, Coovert and McNelis [24] originally proposed to use Petri nets to model team decision making. The first property is that Petri nets are able to model conflicts and simulate concurrency among the components in a system [74]. Figure 20 is an example of Petri net illustrates conflict and concurrency. After T1 fires, P1 becomes empty and P2 and P3 each contains a token. Then T2 and T3 can fire in parallel. After T3 fires, P3 becomes empty and P4 contains a token. There is a conflict between T4 and T5. Either of them needs a token from P4 to fire itself. In this state, only one of them can fire. As one may apply different strategies to decide which will fire, for example randomly choosing one, the firing rule of Petri nets ensure that only one fires.

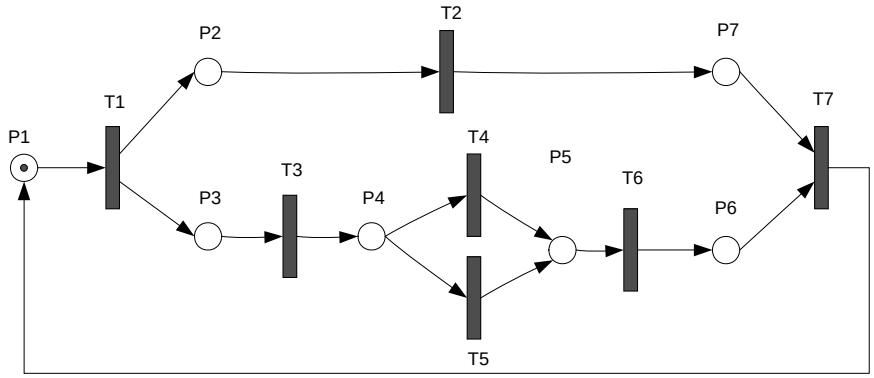

Figure 20. A Petri Net illustrating conflict and concurrency.

The second property is Petri nets are compositional [46]. A system can be divided into various sub-systems and modeled at different levels of abstraction. Subnet components can be used to represent sub-systems in a high level of abstraction. A subnet component at a high level of abstraction (a place, transition or arc) can be expanded by a

subnet at the lower level of abstraction. For example, suppose a sub-system may be represented by a transition. With a low level of abstraction, the sub-system can be represented by a subnet. The transition for the sub-system can be expanded by the subnet for the sub-system. The Petri net in Figure 20 can be modeled as the one in Figure 21 at a high level of abstraction. By expanding the subnet component in Figure 21 by the subnet in Figure 22 we can have the Petri net in Figure 20.

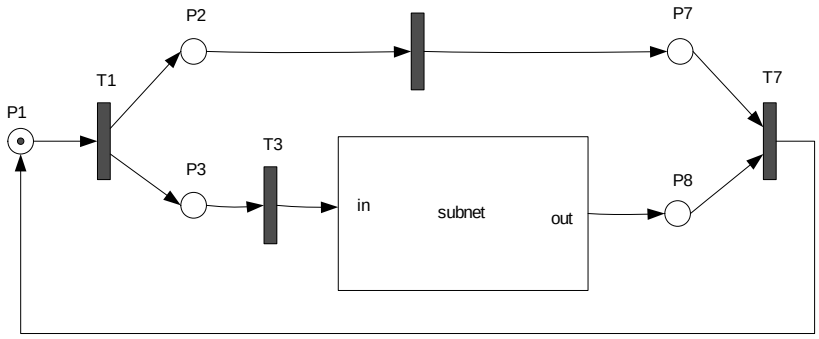

Figure 21. A Petri Net with a subnet component at a high level of abstraction.

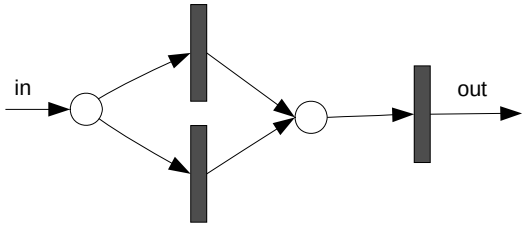

Figure 22. A subnet corresponding to the subnet component in Figure 21.

The third property of Petri nets is that many powerful analyses have been developed, such as reachability, liveness, coverability and decidability, to validate the systems modeled by Petri nets and analyze their behaviors [74]. Considering that Petri nets are compositional and hierarchical, such analyses can be performed at various levels of expansion or reduction entailing varying amounts of time and space.

V.5.2 RoB-CAST Petri Net for Individual Process

While previous versions of CAST used Petri nets, to represent plans, the manner in which they were used is not suitable for RoB-CAST. Moreover, a few extensions make it easier to handle the dynamic composition of responsibility as roles are selected to fulfill role variables. In this section, we will describe an extension of Petri nets, called RoB-CAST-PN, which is developed to represent the knowledge of team process.

V.5.2.1 Overall Ideas

To represent the individual process of an agent only as the portion of a team process related to the agent and to take advantage of the dynamic composition of responsibilities, our RoB-CAST-PN has the following characteristics:

1. Every agent only maintains the portion of team process relevant to itself, an individual process, instead of the whole team process as previous versions did. An individual process is represented by a RoB-CAST-PN. Moreover, an agent is responsible of processing all the transitions in its individual process. We make use of our mechanisms of task decomposition and task delegation (described in Chapter IV) to decide the portion of process relevant to an agent. Every time an individual responsibility is delegated to an agent, the agent translates the individual responsibility graph into a RoB-CAST-PN (which we will show in Sec. V.5.3.2).
2. An individual process (i.e., a RoB-CAST-PN) is compositional. Once an individual responsibility has been delegated to an agent and the agent has translated the individual responsibility graph into a RoB-CAST-PN, the agent dynamically composes its individual process with the RoB-CAST-PN (which we will show in Sec. V.5.3.3).
3. A set of special transitions is used to handle the actions in a team process. We use an operator transition to an invocation of an operator. Also, some actions in a team process are complex actions, such as coordinated operations, plan invocation, role selection, and goal achievement. Special transitions are used to represent the coordination for the complex actions, such as coordination, plan, selection, and goal

transitions. When a special transition fires, a corresponding low level routine will be executed to perform the corresponding processing. More details about transitions will be given in Sec. V.5.2.2.2.

4. As described in Chap. IV, a dependency link represents the temporal order between two actions and the actions could be in two individual responsibility. To ensure that they are performed in accordance with the temporal ordering on them, the temporal ordering needs to be represented in their individual processes properly. We use special places, called proxy places, to help represent temporal relationships among actions in different individual processes. More details about places will be given in Sec. V.5.2.2.1.
5. As a result of adopting proxy places to ensure temporal ordering on actions in different individual process and adopting special transitions to handle complex actions, an individual process can be executed as a regular Petri net is executed in a single platform. Later in Sec. V.5.5, we will present the algorithm for executing an individual process.

V.5.2.2 Places, Transitions and Arcs in RoB-CAST-PNs

Similar to traditional Petri nets, a RoB-CAST-PN also contains a set of places, a set of transitions, and a set of arcs. To realize the methods described in Sec. V.5.2.1, some extensions are made on places, transitions, and arcs.

V.5.2.2.1 Places

Places in a RoB-CAST-PN are similar to places in traditional Petri nets except we add several types of special places, including start, end and proxy places. From the perspective of the execution of a RoB-CAST-PN, i.e., producing tokens and consuming tokens, there is no difference between regular and special places. A place is called a special place by giving a certain special meaning to the place.

A start place represents the starting point of an invocation of an operator, role-based plan, or goal achievement. An end place represents the ending point of an invocation.

When an invocation occurs, a token is put into its start place. When a token is delivered to its end place, the invocation is over.

Proxy places represent connections between two RoB-CAST-PNs (each agent has an individual process represented by a RoB-CAST-PN and thus there are multiple RoB-CAST-PNs for a team of agents) . Every proxy place in an individual process must have a corresponding proxy place in another individual process as the matching element of its pair. One is an in-proxy place and another is an out-proxy place. An out-proxy place can only be an output place of a transition. Once it has a token, it sends the token to its corresponding in-proxy place in another RoB-CAST-PN and removes the token from itself. An in-proxy place can only be an input place of a transition. It only receives tokens from its corresponding out-proxy place in another RoB-CAST-PN. Once an in-proxy place has a token, the transition which input place the in-proxy place is can fire if all of its other input places contain required tokens. If a pair of proxy places is in an individual process (as might happen when an agent assumes multiple roles and role variables), they merge to a regular place. Conversely, a regular place can be split into a pair of proxy places[13]. By using proxy places, temporal ordering on actions in different individual processes can be expressed. For example, suppose T2 and P2 (out-proxy) reside in RoB-CAST-PN PN1 and P2 (in-proxy) and T7 reside in RoB-CAST-PN PN1 in Figure 23. Through proxy place P2, the temporal relationship between T2 and T7 is that T2 fires before T7;

[13] Suppose a pair of proxy places represents a connection between the RoB-CAST-PN for a role and the RoB-CAST-PN for a role variable. When the role is selected to fill the role variable, both RoB-CAST-PNs are used to compose the individual process of the agent to which the role is delegated. So, the pair of proxy places are in an individual process and merges to a regular place. Later when another role is selected to fill the role variable, the agent needs to unload the RoB-CAST-PN for the role variable, including the proxy place for the role variable. But the proxy place for the role still stays. So, the regular place for the pair of proxy places needs to be split.

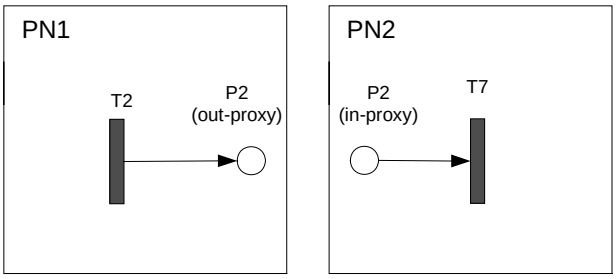

Figure 23. Proxy places in two RoB-CAST-PNs.

V.5.2.2.2 Transitions

Unlike the transitions in traditional Petri net, a transition in RoB-CAST-PN is extended to include a set of conditions and a time delay. The reason why we do so is, a transition representing an operator/plan invocation needs to include the information of the preconditions of the operator/plan. As described in Chapter III, the specification of preconditions includes a set of conditions and a time delay. Moreover, this information affects the decision on whether the transition is enabled. We generalize this extension to all transitions. If a transition does not represent an operator/plan invocation, then its set of conditions is empty and its time delay is 0.

When all input places of a transition contain tokens that are not less than the number labeled on the corresponding arcs to the transition, the attached conditions are checked. If the set of conditions are empty, the transition is enabled. If their conjunction returns true, the transition is enabled; otherwise the transition is not enabled. It is possible that their conjunction returns false initially when the required number of tokens are ready in all input places and turns true later. If this happens before time goes beyond the time delay, the transition is enabled after the condition becomes true; otherwise it is not enabled. The default time delay is infinite.

In addition to the above extensions, a transition can be a special transition by giving a certain special meaning. A special transition could be an operator, plan, selection, coordination or goal transition. When a special transition is fired, a corresponding special routine is called as well as consuming and producing tokens as non-special transitions. We list the special routines for different special transitions in the following:

- If an operator transition fires, the agent performs its corresponding operator. A team operator is decomposed and there is an operator transition in the individual process of every involved agent. The coordination of the team operator is represented by a coordination transition in the individual process of every involved agent, which finally is represented by a subnet. The coordination subnet decides if the operator transition fires.
- If a plan transition fires, the agent coordinates with other involved agents to execute the plan (more exactly, the plan transition coordinates with other corresponding plan transitions in individual processes of other involved agents). First, the agent coordinates with other involved agents to find an admissible team assignment. To execute the plan, the plan transition is expanded by a subnet of RoB-CAST-PN corresponding to the individual responsibility of the role(s) delegated to the agent according to the admissible team assignment.
- If a selection transition fires, the agent coordinates with other involved agents to run Algorithm 14 for role selection in Chapter IV and decides which role (consequently the agent to which the selected role is delegated) is selected.
- If a coordination transition fires, the agent coordinates with other involved agents and together they decide which agent(s) will continue to execute its (their) action(s) which is (are) under the coordination. If the coordination is AND, then all agents will execute their actions which are under the coordination. If the coordination is OR, at least one agent executes its action(s) which is (are) under the coordination. If the coordination is XOR, exactly one agent executes its action(s) which is (are) under the coordination. After coordinating with other agents, if an agent executes its action(s) which is (are) under the coordination, then a token is put in the place to each output place of the coordination transition.
- If a goal transition fires, the agent coordinates other involved agents to invoke some planning algorithm for finding a course of actions to achieve their goal.

V.5.2.2.3 Arcs

Arcs in RoB-CAST-PN are similar the arcs in traditional Petri nets except a special type of arc is include. The special type of arc, called a guard arc, is used to represent condition evaluations for flow control. Unlike a normal arc labeled with a constant number, a guard arc is labeled with a condition. For our purpose of representing a condition evaluation, a guard arc only stems from a transition to a place. When the transition fires, the condition labeled with the guard arc is checked. If it is true, a token is put in the place. Suppose there is a condition evaluation *c*. It can be represented as Figure 24. T1 is a transition with an empty set of conditions and a time delay 0 for the purpose of flow control. G1 and G2 are two guards and are labeled with condition *c* and $\neg c$ respectively. When T1 fires, if *c* is true, then a token is put in P2; otherwise, a token is put in P3.

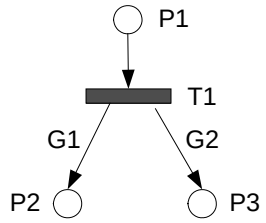

Figure 24. Guard arcs representing a condition evaluation.

Although RoB-CAST-PN is not directly an extension of Predicate/Transition (PrT), RoB-CAST-PN can easily translated into a PrT net. In a PrT net, every place is a predicate. The places in RoB-CAST-PN is not represented by a predicate directly. Places in RoB-CAST-PN are used to connect transitions representing certain actions which transit the environment from a state to another state. Places essentially represent such states and the states can be represented by predicates. Hence, a RoB-CAST-PN net can be easily translated to a PrT net, and then various analysis facilities of PrT nets can be applied to RoB-CAST-PN nets.

V.5.2.3 Definition of RoB-CAST-PN

With the above extensions, the RoB-CAST-PN[14] is defined as a tuple, N = (P, T, C, D, ϕ, F, G, L, M_0, T_e, T_x, T_s, N_s, S), where:

1. P is a finite set of places. P contains special places, including start, end and proxy places;
2. T is a finite set of transitions. $P \cap T = \emptyset$ and $P \cup T \neq \emptyset$. T contains special transitions, including operator, plan, selection, coordination and goal transitions. Special transitions also record information for corresponding processing. An operator transition records the name, arguments, preconditions and effects of the corresponding operator. A plan transition records the name, arguments, preconditions, effects, and termination conditions of the corresponding plan, and the names of the roles involved. A selection transition records the name of the corresponding role variable, the names of the roles in the selection scope, and the select constraints. A coordination transition records the coordination mode (i.e., AND, OR, or XOR) and the transitions in other RoB-CAST-PN with which the coordination node coordinates. A goal transition records the goal to be achieved and the names of the involved roles.
3. C is a finite set of conditions in first order formula (conjunction of predicates). $\emptyset \in C$ and its evaluation always returns true;
4. D is the set of constants; Infinity ∞ is a special constant in D.
5. ϕ is a mapping function on T. $\forall t \in T$, $\phi(t) = (c, d)$, where $c \in C$ and $d \in D$;
6. F is a finite set of arcs and $F \subseteq (P \times T) \cup (T \times P)$;
7. G is a finite set of guard arcs. Every guard arc is labeled with a condition $c \in C$. $G \subseteq F$ and more exactly $G \subseteq T \times P$. The condition labeled with a guard arc (t, p) is denoted by G(t, p);

[14] As described earlier, an individual process of an agent is represented by a RoB-CAST-PN. An individual responsibility will be translated to a RoB-CAST-PN. When an individual responsibility is delegated to an agent, the agent dynamically composes the RoB-CAST-PN of its individual process with the RoB-CAST-PN for the individual responsibility.

8. L is a labeling function on F. $\forall$ f$\in$F, L(f) $\in$D. Although L(f) could be any non-negative integer, all arcs currently in our teamwork architecture are labeled 1 because passing one token is enough for the flow controls represented by all kinds of MALLET constructs. We use L(t, p) or L(p, t) to represent the label on the arc from t to p or from p to t;
9. M0 is the initial marking. M0 = $\underset{p\in P}{\mathbf{Y}}$ M0(p), where M0(p) is the set of tokens residing in place p. $\underset{p\in P}{\mathbf{Y}}$ M0(p) represents a n-vector, where n = |P| and each element is the number of tokens corresponding to a place in P;
10. Te is a set of entrance transitions. Te $\subseteq$ T and Te could be empty or contain a transition;
11. Tx is a set of exit transitions. Tx $\subseteq$ T and Tx could be empty or contain a transition;
12. Ts is a set of transitions that can be replaced by RoB-CAST-PNs;
13. Ns is a set of RoB-CAST-PNs;
14. S is a set of transition-subnet substitutions (t', N'), where t'$\in$ Ts, N' = = (P', T', C', D', ϕ', F', G', L', M0', Te', Tx', Ts', Ns', S'), and N' $\in$Ns. To expand transition t' with subnet N', t' is removed from N; all places in P', all transitions in T' and all arcs in F' are added into N; the input arcs of t' are changed to point to the only transition in Te' if the transition exists or removed from N otherwise; and the output arcs of t' are changed to stem from the only transition in Tx' if the transition exists or removed from N otherwise. To reduce a subnet N' with transition t', the reverse procedure restore N back to what N was before t' expands.

For easier description, we introduce some additional notions. Let $\cdot t$ = {$p\in$P| (p, t) $\in$F) be the set of input places of transition t and $t\cdot$ = {$p\in$P| (t, p) $\in$F) the set of output place of transition t. A marking M is a function from the set of places P to the nonnegative integers I, denoted by M: P$\rightarrow$I. M(p) denotes the number of tokens in place p under marking M. M_0 described above is a special marking for a RoB-CAST-PN in

the initial state. Let $gv(t, p)$ be the guard value of an arc from transition t to place p. $gv(t, p)$ is defined by

$$gv(t,p) = \begin{cases} L(t,p), & (t,p) \in F \wedge (t,p) \notin G \\ value(G(t,p)), & (t,p) \in G \end{cases}$$

where $value(G(t, p))$ is 1 if the guard condition G(t, p) is true and 0 otherwise.

We note that our RoB-CAST-PN is not an extension of Timed Petri Nets even though we attached time with the conditions of transitions. Timed Petri Nets typically attach time with places and transitions to respectively represent when the tokens in the places are available for enabling transitions and how long the firings of transitions take. As our RoB-MALLET cannot represent teamwork knowledge related to time (except that time could be associated with preconditions. As discussed in the definition of RoB-CAST-PN, representing such time information does not require RoB-CAST-PN to be a Timed Petri net), our RoB-CAST-PN does not include the expressivity of Timed Petri Nets. In Chapter VII, we will have more discussions about including expressivity of Timed Petri Nets into RoB-CAST-PN in future work.

V.5.3 Maintenance of Individual Process

As a model representing process knowledge, the individual process of an agent decides its behaviors, i.e., what actions next. Moreover, the individual process evolves with its execution; particularly it dynamically expands and reduces with plan invocations. Benefiting from role delegation in Chap. IV, team activities are decomposed into individual responsibilities of roles and then the roles are delegated to agents. To actually execute the actions in the individual responsibilities, each agent needs to translate the responsibilities of roles delegated to it into the format of RoB-CAST PN. The individual process is thus dynamically compositional. A role delegated to an agent may involve a sub-plan invocation (say P_1). For example, to execute sub-plan P_1, some role (say r_1) in the sub-plan is delegated to an agent and the plan transition for the sub-plan invocation is dynamically expanded by the RoB-CAST PN representation of the responsibility of role r_1. Moreover, after P_1 is done, the RoB-CAST PN

representation of the responsibility of role r_1 is reduced to the plan transition. The reason why we do this is, after P1 is done, the transitions in P1 cannot be enabled unless the plan transition is expanded again; reducing back to the plan transition avoids unnecessary checking of whether those transitions are enabled.

In the following sections, we will demonstrate how an individual process represents the initialization and finalization of an invocation, how to translate individual responsibility into RoB-CAST-PN representation, and how agents dynamically compose their individual processes.

V.5.3.1 Initializing and Finalizing an Invocation

An invocation can be started by a (sub)team of agents by the START construct described in Chap. III. To actually execute the invocation, the process knowledge of the agents must reflect it.

Suppose team T1 containing agents ag_1, ag_2, …, ag_n starts a plan Plan1 by *(START T1 (Plan1))*. To initialize the invocation of P1 in the individual process of each agent in T1, a RoB-CAST PN is added for each agent as shown in Figure 25. P1 is a start place. P2 is an end place. T1 is a plan transition in which plan name `Plan1` and team name T1 are attached. $\phi(T1)$ is (c, d), where c is the precondition of `Plan1` and d is its evaluation duration (both c and d are specified in the plan definition of `Plan1`). If the invocation is an operator or a goal, instead of a plan, T1 is an operator or goal transition correspondingly. Initially, a token is added to P1 so that T1 can be enabled if c is satisfied within duration d.

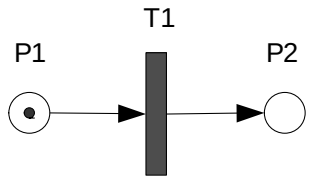

Figure 25. A RoB-CAST-PN for initializing an invocation.

The plan `Plan1` can be started many times by multiple START statements, perhaps by different teams of agents. An agent may participate in multiple instances of invocation of Plan1. Also, plan transition T1 needs to coordinate with other corresponding plan transitions in other individual processes. It is necessary to make sure those plan transitions reflect the same plan invocation. For these reason, P1, P2, and T1 must be named by a global convention.

We assume that there is a centralized name server and it has a clock. The name server makes sure that no two invocations can start exactly at the exactly same time. The name conventions for P1, P2 and T1 are based on the name server clock as follows:

Name of P1 = start time of the invocation + ".Start"

Name of P2 = start time of the invocation + ".End"

Name of T1 = start time of the invocation

Although there may be other implementation to input a START statement to agents, a START statement in our teamwork architecture is input through an agent, which initiates the other agents in the team. This implementation allows us to dynamically start an invocation of an operator/plan/goal. This agent coordinates all involved agents to add the invocation into their individual processes.

Every agent executes its individual process and fires T1 in its individual process. When T1 fires, a special routine is called to synchronize other involved agents to perform corresponding processing. Suppose T1 is a plan transition for invoking `Plan1`. When each agent fires T1, it calls the routine for plan invocation (described earlier in Sec. V.5.2.2.2), which coordinates with all other involved agents to decide an admissible team assignment and then expand T1 with a RoB-CAST-PN for the responsibility of the role delegated to the agent. The agents actually execute the plan by each agent executing the RoB-CAST-PN for the responsibility of the role delegated to the agent.

Once the RoB-CAST-PN is complete, a token has been delivered to P2 and the plan invocation is over. To finalize the plan invocation, P1, P2 and T1 are removed from the individual process. If we did not recycle the RoB-CAST-PN components for completed invocations, then the size of the RoB-CAST-PN of the individual process would become

larger and larger with the execution of the individual process. Given those RoB-CAST-PN components are no longer useful (we call them expired RoB-CAST-PN components), memory leakage would result. Moreover, time performance would be decreased. Every time the agent searched for enabled transitions, it would need to check those expired transitions. For this reason, expired RoB-CAST-PN components are recycled. RoB-CAST-PN supports recycling them because of its dynamic composition (described later in Sec. V.5.3.3). In contrast, CAST 3.0 does not do this and hence has to check all the extra transitions.

V.5.3.2 Translating Individual Responsibility into RoB-CAST-PN

When a plan transition fires, some role(s) in the plan is(are) delegated to the agent (described in Sec. IV.5) via the process of firing a plan transition (described in Sec. V.5.2). As explained in Sec. IV.4, the agent is obligated[15] to perform the actions associated with the role(s) which are captured by the graphical representation of responsibility. To add those actions into the individual process of the agent, the responsibility of the role(s) delegated to the agent needs to be translated into a RoB-CAST-PN representation.

One may wonder if the responsibility graph and the RoB-CAST-PN representation can be merged into one or just use one of them. Two reasons drove us to have separate representations for responsibilities and individual processes:

- Our responsibility graph language is suitable for partitioning team activities into individual activities. The representation of responsibility has coarser granularity

[15] As explained in Chapter IV, if a role is delegated to an agent or the role delegated the agent is selected to fill a role variable, the agent forms an intention to execute the actions associated with the role or role variable. The search of an admissible team assignment and a role selection ensures that the agent is capable to execute the actions. Moreover, the agent's intention is relative to the termination conditions. The agent may drop its intention once the termination conditions are satisfied. For example, a vehicle agent cannot move if it runs out of fuel. This can be represented by termination conditions, and the vehicle agent drops its intention to execution actions once it runs out of fuel. We note that, people could develop some mechanisms to handle such failures, for example delegate to other agents as discussed in Ioerger's and Johnson's responsibility [43]. In RoB-CAST, no such mechanism is not enabled and agents just dismiss their intention to execute their actions. However, RoB-CAST enable proactive helping behaviors (discussed later in Sec. V.6.3), other agents may sense such failure and provide helps to the agents.

than RoB-CAST-PN. The actions and temporal orders described by RoB-MALLET are directly represented by nodes and links in the representation of responsibility. However, they need to be represented by sets of places, transitions and arcs in RoB-CAST-PN, such as a condition evaluation for a branch or loop statement. Therefore, it is much easier to partition the responsibility represented by the responsibility graph language than that represented by RoB-CAST-PN.

- The responsibility graph language and RoB-CAST-PN have different focuses. The responsibility graph language emphasizes what actions individuals in a plan have and the temporal orders on them. RoB-CAST-PN emphasizes the execution of a plan, for example, globally unique naming and dynamic composition, as described in the next sub-section.

In the following sub-sections, we will describe our overall procedure for translating an individual responsibility to a RoB-CAST-PN. The overall procedure basically translates the nodes in the individual responsibility into transitions first and then connects the transitions according to the links connecting the nodes in the individual responsibility. We will explain how to translate nodes into transitions and how to connect the transitions. We will also describe the naming convention during the procedure. At last, we will present an algorithm that implements the procedure for translating an individual responsibility to a RoB-CAST-PN.

V.5.3.2.1 Overall Procedure

The representation of the responsibility of an individual (a role or role variable) contains a set of nodes and a set of directed links connected to two nodes. Both nodes and links represent information relevant to actions associated with the individual. The information should be translated into the format of RoB-CAST-PN. So all nodes and links in a responsibility should be translated into some elements in RoB-CAST-PN. To translate a responsibility to a RoB-CAST-PN representation, we adopt the following overall procedure (later in Sec. V.5.3.2.3, we will give a set of figures to illustrate the details in the procedure, particularly backward/forward chaining to connected places and transitions generated in this procedure):

1. Translate all nodes. A node in the responsibility (described in Sec. IV.4.2) is translated into a corresponding transition;
2. For each input link (which is not a plan, coordination, selection or goal link) of each node, create a place as an input place of the transition corresponding to the node, and connect the place to the transition. This step is called backward chaining.
3. For each output link (which is not a plan, coordination, selection or goal link) of each node, connect the transition corresponding to the node to the place corresponding to the link (an output link is also an input link of the node to which the link points), which is created in (2). This step is called forward chaining.

In the above overall procedure, connecting transitions is realized by (2) and (3). For a link connecting node n1 to node n2, to connect transition t1 (corresponding to node n1) and transition t2 (corresponding to node n2), backward chaining creates a place between t1 and t2 and connects t2 to the place; and forward chaining connects t1 to the place. To connect a transition and a place, a set of transitions, arcs and/or places may be used for different scenarios. In Sec. V.5.3.2.3, more details will be given to show how backward and forward chaining connect transitions and places.

V.5.3.2.2 Translating Nodes to Transitions

Translating a node in a responsibility graph is quite straightforward. It basically translates a node to its corresponding transition. An operator node is translated into an operator transition. A plan node is translated into a plan transition. A selection node is translated into a selection transition. A goal node is translated to goal transition. A connector node is translated into a regular transition. A coordination node is translated in to a coordination transition.

When translating plan/coordination/selection/goal nodes, their corresponding plan/coordination/selection/goal links are translated into coordination information for the transitions corresponding to their source nodes. For example, plan links are translated to

coordination information of their corresponding plan transitions (i.e., with which individuals the agent coordinates to decide the team assignment for invoking their plan).

For an operator/plan node, the precondition of its corresponding operator/plan is translated into the mapping function ϕ. For other types of transitions, their mapping function ϕ is (Ø, 0). For a plan/selection/goal/coordination node, its corresponding transition also attaches the information carried by its corresponding links (i.e, the information about coordinating with which transitions in the RoB-CAST-PNs of which individuals).

Later in Sec. V.5.3.2.5 when we present algorithm for translating an individual responsibility into a RoB-CAST-PN *PN*, we use Translate_Node(*node*, *PN*) to denote the translation of a node into its corresponding transition as described above. Translate_Node(*node*, *PN*) also add the transiton to a RoB-CAST-PN *PN*.

V.5.3.2.3 Connecting Transitions

Transitions are connected by interpreting dependency/guard links between the nodes corresponding to the transitions. A dependency/guard link L connecting node N1 to node N2 represents a temporal order between the actions corresponding to N1 and N2. As described above, N1 and N2 are translated into their corresponding transitions. The link L is translated by backward and forward chaining. Several factors affect backward and forward chaining, include whether a link is an inter-link, what type a link is (dependency or guard), what type a node is (i.e., for what type of flow control). In the following, we describe, with a set of figures, how backward and forward chaining connect transitions and places for all possible scenarios. A figure may include more than one scenario, so a figure may be used in more than one item below.

- Backward chaining a non branch-end/condition-with-loop-back node. In Figure 26, a dependency or guard link L1 connects N1 to N2 (e.g., (Seq (Do r1 (op1)) (Do r1 (op2))) is represented by L1, N1 and N2, where N1 represents op1, N2 represents op2, and L1 connects N1 and N2 and represents that op1 is executed before op2). N1 and N2 are translated to T1 and T2. The back chaining of N2 with L1 creates a place P1 and an arc connecting P1 to T2;

- Backward chaining a branch-end node. In Figure 27, a branch-end node BE is translated into transition T2. The backward chaining of BE with L3 creates a transition T3, a place P4, and an arc connecting them. P4 is prepared for the forward chaining of the predecessor node of L3. The backward chaining of BE with L4 creates a transition T4, a place P5, and an arc connect them. P5 is prepared for the forward chaining of the predecessor node of L4. A place P3 and three arcs are also created to connect T2, T3, and T4. For easier description, we call P3 a branch-end place of node BE; and we call T3 a backward chaining transition of link L3 and T4 a backward chaining transition of link L4;
- Backward chaining a condition node with a loop-back input link. In Figure 28, a condition node C1 has a loop-back link L4 and a link L1 which initially enter a loop. C1 is translated into T1. The backward chaining of C1 with L4 creates a place P2, a transition T2, and an arc to connect them. P5 is for the forward chaining of the predecessor of L4. The backward chaining of C1 with L1 creates a place P5, a transition T4, and an arc to connect them. P5 is for the forward chaining of the predecessor of L1. A place P1 and three arcs are also created to connect T1, T2, and T4. For easier description, we call a place, which serves the same purpose as P1, a loop-start place; and we call T4 a backward chaining transition of link L1 and T2 a backward chaining transition of link L4;
- Forward chaining a non-condition node. In Figure 26, the forward chaining of N1 with L2 creates an arc (say Arc1) connects T1 and P1;
- Forward chaining a condition node. In Figure 27, C1 is a condition node with condition *c*; L1 is a true guard link; and L2 is a false guard link. C1 is translated into T1. Suppose that P1 is created by the backward chaining of the node that L1 connects to and that P2 is created by the backward chaining of the node that L2 connects to. The forward chaining of C1 with L1 creates an arc (say Arc1) to connect T1 and P1 and sets Arc1 as a guard arc with condition $G(\text{T1}, \text{P1}) = c$. The forward chaining of C1 with L2 creates an arc (say Arc2) to connect T1 and

P2 and labels Arc1 with $G(\text{T1, P1}) = \neg c$. The forward chaining of C1 in Figure 28 is similarly illustrated;

- Backward and forward chaining an inter-link. In Figure 29, N1 and N2 belong to different individual responsibilities. L1 connects N1 and N2. N1 and N2 are translated into T1 and T2 respectively. To represent L1 in the format of RoB-CAST-PN, the backward chaining of L1 prepares an in-proxy place instead of a regular place; and the forward chaining of L1 creates a corresponding out-proxy place and an arc to connect T1 to it.

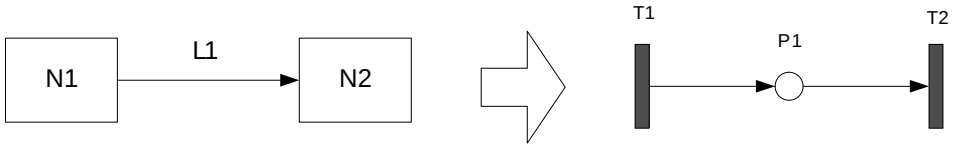

Figure 26. Case 1 of backward and forward chaining.

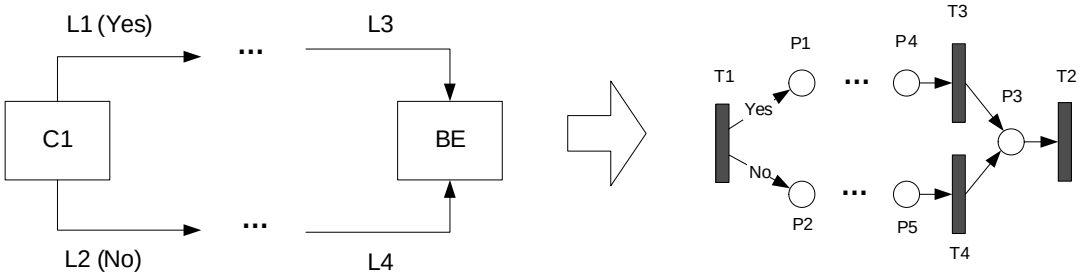

Figure 27. Case 2 of backward and forward chaining.

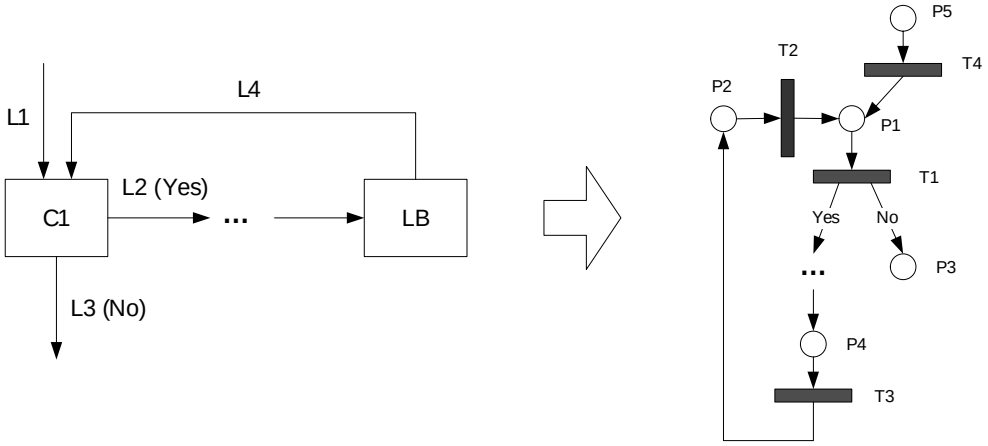

Figure 28. Case 3 of backward and forward chaining.

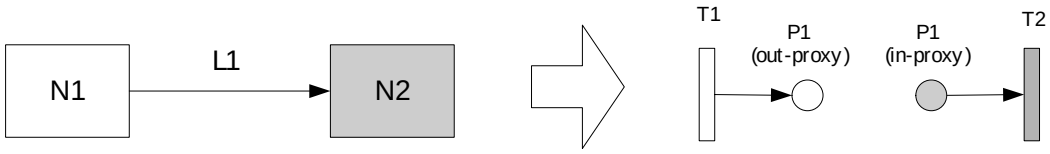

Figure 29. Proxy places for chaining an inter-link.

V.5.3.2.4 Naming Convention

To avoid naming conflicts, we also define a naming convention for places, transitions and arcs generated. As the result of the naming convention, there is no naming conflict for different instances of an invocation of the same plan; the names of special transitions (plan, goal, and selection transitions) for a complex action match among the RoB-CAST-PNs corresponding to all individual responsibilities so that coordination implied by them can occur correctly; and the pairs of proxy places among RoB-CAST-PNs should match so that token passing occur correctly.

When generating team responsibility, plan/coordination/selection/goal nodes for different roles in a same corresponding statement are named differently. However, they reflect the same coordination. For easily finding the corresponding transitions in the RoB-CAST-PNs for the individual responsibilities of the different roles, we unify the

names of all corresponding nodes and then name the the corresponding transitions based on the unified name. Suppose there is a node *n* for a complex action (i.e., a plan/coordination/selection/goal node). We use *UnifiedName*(*n*) to denote its unified name of all corresponding nodes. Node *n* has a link to each of its corresponding nodes, so it is easy to find their names. *UnifiedName*(*n*) can easily be resolved by following a protocol based on these names, for example, their minimum in alphabetic order. For easier description, we extend *UnifiedName*(*n*) to all kinds of nodes. If *n* is not a plan/selection/goal node, *UnifiedName*(*n*) is just the name of *n*.

Suppose a plan transition of plan P is named T. We use the following naming convention to globally name an individual responsibility of a role or role variable in P:

Transition of node n = T + "." + UnifiedName(n)

Place backward chaining link l = T + "." + l + ".P"

Transition of backward chaining l = T + "." + l + ".T"

Branch-end place of n = T + "." + UnifiedName(n) + "." + "BE"

Loop-start place of n = T + "." + UnifiedName(n) + "." + "LS"

Arc = [source] + "_" + [target]

In the above formulas, *[source]* means the name of source place/transition of an arc; and *[target]* means the name of target place/transition. For example, suppose there are a plan transition T (for invoking plan p1) with a name "12:30:35.p1" and a node n with a unified name "N11" in the responsibility of plan p1. Following the name convention, the transition corresponding to node n is named "12:30:35.p1.N11".

V.5.3.2.5 Algorithm of Translating Individual responsibility Into RoB-CAST-PN

We have developed an algorithm which implements the methods described in previous sections to translate an individual responsibility into a RoB-CAST-PN. To simplify the description of the algorithm, any RoB-CAST-PN element created in the algorithm is added into the RoB-CAST-PN on which the algorithm is working. Every new RoB-CAST-PN element is named by the above naming convention. Also, all non-guard arcs are labeled 1. The following is the algorithm of translating an individual responsibility into a RoB-CAST-PN:

Algorithm 15. Translating an individual responsibility into a RoB-CAST-PN

```
Translate_Responsibility(Resp)
  Begin
    Create a RoB-CAST-PN PN;
    For each node n in Resp do
      t = Translate_Node(n, PN);
      If n is a start node
        Set n as the Entrance of PN;
      ElseIf n is an end node
        Set n as the Exit of PN;
      End-If;
      Backward-Chaining(n, t, PN);
    End-For;

    For each node n in Resp do
      Let t be the transition corresponding to n;
      Forward-Chaining(n, t, PN);
    End-For;
    Return PN;
  End

Backward-Chaining(n, t, PN)
  Begin
    If n is a Branch-End node
      Create a Branch-End place bep;
      Create an arc to connect bep to t;
      For each input link l of n do
        If l is an interlink
          Create an in-proxy place bp;
```

```
            Else
              Create a place bp;
            End-If;
            Create a transition bt;
            Create an arc to connect bp to bt;
            Create an arc to connect bt to bep;
          End-For;
        ElseIf n is a condition node with a loop-back input link
          Create a Loop-Start place lsp;
          Create an arc to connect lsp to t;
          For each input link l of n do
            If l is an interlink
              Create an in-proxy place bp, for backward chaining n;
            Else
              Create a place backward chaining l, bp;
            End-If;
            Create a transition backward chaining n, bt;
            Create an arc to connect bp to bt;
            Create an arc to connect bt to lsp;
          End-For;
        Else
          Create a place. bp, for backward chaining l ;
          Create an arc to connect bp to t;
        End-If;
      End

Forward-Chaining(n, t, PN)
    Begin
      For each input link l of n do
```

If *l* is an interlink

Create an out-proxy place *fp*;

Else

Let *fp* be the place created when Backward-Chaining(*n*, *t*, *PN*) on link *l*;

End-If;

Create an arc, *arc*, to connect *t* to *fp*;

If *n* is a condition node with condition *c* and *l* is a guard link

If *l* is a true guard

Set the guard condition of *arc* as *c*;

Else

Set the guard condition of *arc* as $\neg c$;

End-If;

End-If;

End-For;

End

By this algorithm, the individual responsibility of role r1 in Figure 22 is translated into the RoB-CAST-PN in Figure 30, where T1 and T8 represent the entrance and exit nodes; T2 represents the condition node for condition `(eq 1 1)`, and G1 and G2 are two guard arcs with condition `(eq 1 1)` and its negation respectively; T4 represents the operator node for operator `selectTarget`; T6 represents the plan node for plan `senseandmove`; T5 represents the condition node for condition `(targetloc r1 ?X1 ?Y1)`, G3 and G4 are two guard arcs with condition `(targetloc r1 ?X1 ?Y1)` and its negation respectively, P7 is an out-proxy place, P9 is an in-proxy place, and their peers are in the RoB-CAST-PN translated from the individual responsibility of role r2. The individual responsibility of role r2 in Figure 22 is translated into the RoB-CAST-PN in Figure 31, where T10 represents the condition node for condition `(eq 1 1)`, and G5 and G6 are two guard arcs with condition `(eq 1 1)` and its negation respectively; T11 represents the operator node for operator `waitGold`; T13 represents the plan node for plan `moveto`; T12 represents the condition node for condition

`(targetloc r2 ?X2 ?Y2)`, and G7 and G8 are two guard arcs with condition `(targetloc r2 ?X2 ?Y2)` and its negation respectively, P7 is an in-proxy place, P9 is an out-proxy place, and their peers are in the RoB-CAST-PN translated from the individual responsibility of role r1.

We note that the translation of individual responsibilities into RoB-CAST-PNs is faithful to the semantics of individual responsibilities. As explained in Chapter IV, a role-based plan is translated into a team responsibility graph and then the team responsibility graph is decomposed into individual responsibilities. So, the RoB-CAST-PNs are faithful to the role-based plan. If there is any deadlock in the role-based plan, the RoB-CAST-PNs will contain a deadlock correspondingly. If the role-based plan does not have a deadlock, neither will the RoB-CAST-PNs. The analysis of liveness of Petri Nets can be applied to detect deadlocks in role-based plans.

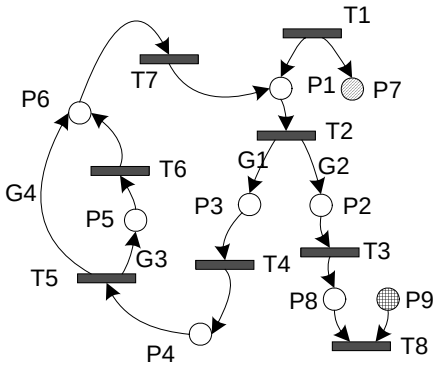

Figure 30. The RoB-CAST-PN translated from the individual responsibility of r1 in Figure 22.

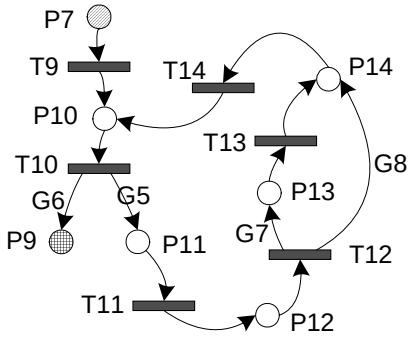

Figure 31. The RoB-CAST-PN translated from the individual responsibility of r2 in Figure 22.

V.5.3.3 Dynamic Composition

When an invocation is started by a team of agents, each agent initializes the invocation as described in Sec. V.5.3.1.

If the invocation is an operator, its corresponding operator transition can fire directly without any expansion.

If the invocation is a plan invocation, each agent fires its corresponding plan transition and calls the corresponding routine (described in Sec. V.5.2.2.2) so as to coordinate with other involved agents to decide an admissible team assignment, which will be recorded in the agent's team organization (described in Sec. V.4). Based on the admissible team assignment, each agent knows which role(s) are delegated to it and determines the individual responsibility of the role(s).

To actually execute the plan, the plan transition of the agent should be expanded by the RoB-CAST-PN representation of the responsibility of the role(s). As explained in Sec. IV.4, all actions associated with a role can be represented by its individual responsibility. In the last section, we have explained how to translate a responsibility into RoB-CAST-PN representation. Therefore, the plan transition should be expanded by the RoB-CAST-PN representation translated from the responsibility of the role(s) delegated to the agent.

Moreover, a plan invocation contains various sub-invocations, including operator, plan, and goal invocation and role variable selections. They correspond to operator, plan, goal, and selection transitions. Except selection transitions, operator/plan/goal transitions can be repeatedly handled as we explained above. For a selection transition, if the role delegated to the agent is selected to fill the role variable, the responsibility of the role variable is translated into the RoB-CAST-PN subnet. The selection transition itself only represents the coordination about the agent involved in selecting roles to fill a role variable. However, unlike a plan transition representing the actions associated with the roles in the plan delegated to the agent, the actions associated with the role variable have not been reflected in the individual process. If a role delegated to the agent is selected to fill the role variable, the actions associated with the role variables need to be represented

in the individual process. Therefore, the agent just loads the RoB-CAST-PN representation for the individual responsibility of the role variable into its individual process. This can be treated as an empty transition is expanded by a subnet.

If the invocation is to achieve a goal, the agent fires its corresponding goal transition. The agent calls the corresponding routine (described in Sec. V.5.2.2.2) so as to coordinate with other involved agents to find a course of actions for the goal. The course of actions can be treated as a plan. So, the goal invocation turns to be a plan invocation and the goal transition can be replaced by a plan transition.

Now we explain how a transition is expanded by a subnet. Suppose the individual process of agent *ag* is IP = (P, T, C, D, ϕ, F, G, L, M_0, T_e, T_x, T_s, N_s, S), *t* is a transition (plan transition or empty transition for role variable selection) to be expanded, and the subnet to expand *t* is PN1 = (P1, T1, C1, D1, $\phi 1$, F1, G1, L1, $M_0$1, T_e1, T_x1, T_s1, N_s1, S1). We use IP′ = (P′, T′, C′, D′, ϕ', F′, G′, L′, M_0′, T_e′, T_x′, T_s′, N_s′, S′) to denote the individual process after *t* is expanded by PN1. A transition *t* in IP is expanded by PN1 via the following procedure:

1. P′ = P $\cup$ P1. All places in P and P1 are included in P′. If a matching pair of proxy places is included, they merge into a regular place with same name;
2. T′ = T $\cup$ T1 $-\{t\}$
3. C′ = C $\cup$ C1
4. D′ = D $\cup$ D1
5. $\phi' = \phi \cup \phi 1 - \{\phi(t)\}$
6. F′ = F $\cup$ F1. However, the arcs in F to *t* are changed to the only one transition in T_e1 if T_e1 is not empty or removed otherwise; and the arcs in F from *t* are changed to the only transition in T_x1 or removed otherwise[16];
7. G′ = G $\cup$ G1. If an arc, *arc*, is removed from F in step (6), remove G(*arc*);

[16] When generating team responsibility for a role-based plan, only one start node and one end node are generated. After partitioning team responsibility into individual responsibilities, each of them just resides in the individual responsibility of one role. When translating individual responsibilities into RoB-CAST-PN representations, they are translated into one entrance transition and one exit transition respectively. So, exactly one agent has the entrance transition and one has the exit transition. If the agent does not takes the responsibility with the start or end node, T_e1 or T_x1 will be empty. It is possible for an agent to have the entrance transition, the exit transition, both of them, or none of them.

8. $L' = L \cup L1$. If an arc, *arc*, is removed from F in step (6), remove L(*arc*);
9. M_0' grows from M_0 according to the growth of P′ from P, however, the marking of IP does not change. $\forall p \in P' \cap P$, $M_0'(p) = M_0(p)$ and $M_0'(p) = 0$ otherwise. Actually, PN1 is translated from the responsibility of role *r*. It contains no tokens.
10. $T_e' = T_e$, and $T_x' = T_x$. Expanding a transition in a RoB-CAST-PN does not affect itself to be a sub-net to expand other transition;
11. $T_s' = T_s + \{t\}$;
12. $N_s' = N_s \cup \{PN1\}$;
13. $S' = S \cup \{(t, PN1)\}$. Add this expansion into the transition-subnet substitution set.

Suppose there is an agent ag1. Its individual process is shown in Figure 32. The transition T02 is for invoking plan `scanandcollect` described in Sec. IV.4.3.10. And, r1 in plan `scanandcollect` is delegated to agent ag1. The RoB-CAST-PN translated from the individual responsibility of r1 is shown in Figure 30. When T02 fires, agent ag1 dynamically expands T02 with the RoB-CAST-PN shown in Figure 30. After agent ag1 expands T02, agent ag1's individual process is shown in Figure 33.

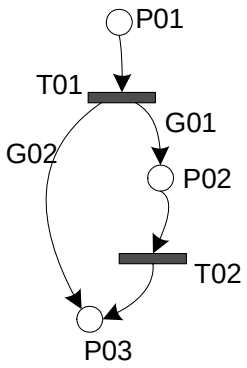

Figure 32. Agent ag1's individual process before expanding T02.

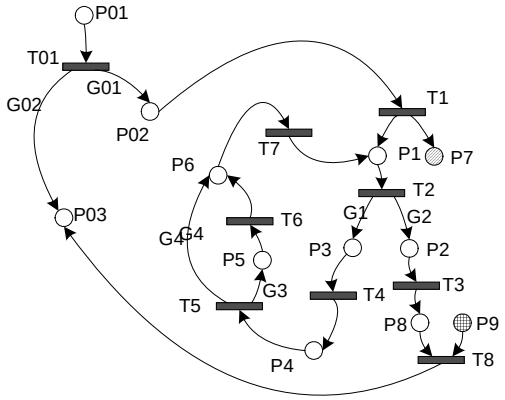

Figure 33. Agent ag1's individual process after expanding T2'.

As another example, suppose agent ag2 is involved in a role selection for a role variable *?rv* and the role *r* delegated to agent ag2 is selected. Agent ag2's individual process before *r* is selected to fill *?rv* is shown in Figure 34. In Figure 34, T4 represents that agent ag2 is involved the role selection. The RoB-CAST-PN for the individual responsibility of the role variable *?rv* is shown in Figure 35. There are two pairs of proxy places in agent ag2's individual process and the RoB-CAST-PN for the individual responsibility of the role variable *?rv*. Once the role r is selected, ag2 appends The RoB-CAST-PN for the individual responsibility of the role variable *?rv* to its individual process. Agent ag2's individual process after *r* is selected to fill *?rv* is shown in Figure 36. We note that those two pairs of proxy places merge to two regular places (P4 and P6) in Figure 36.

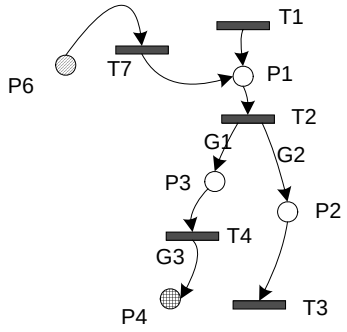

Figure 34. Agent ag2's individual process before *r* is selected to fill *?rv*.

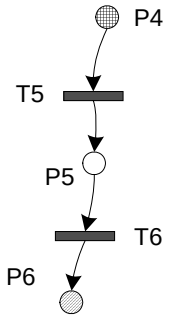

Figure 35. The RoB-CAST-PN for the individual responsibility of the role variable *?rv*.

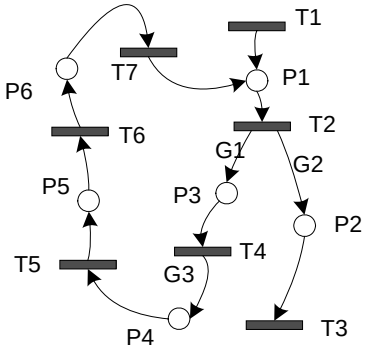

Figure 36. Agent ag2's individual process after *r* is selected to fill *?rv*.

Once a plan invocation is completed RoB-CAST-PN subnet can shrink back to its corresponding plan transition. Once another role delegated to another agent is selected when a role selection fires again, the RoB-CAST-PN for the role variable can be removed. These can be realized just by reversing the above procedure. S´ keeps track of all expansions, so the required information to reverse the above procedure is available.

V.5.4 Interaction with Team Organization

Individual processes and team organizations are two parts of a team process. The individual process of an agent maintains the portion of team process related to the agent. Team organization maintains the shared goals of a team of agents and their joint intentions to achieve the goals. To execute a team process, an individual process and a team organization interact with each other: the team organization is referred to by the individual processes to ensure that they execute correctly; the execution of the individual

process updates the team organization, and further drives dynamic composition of the individual process.

An individual process refers to the team organization to ensure that it is executed correctly in two situations.

- The first is to guarantee the temporal order between two actions of two roles. This situation appears as two transitions, say T1 and T2 (suppose T1 fires before T2), reside in two individual processes and are connected by a matching pair of proxy places. If T1 fires, a token is put in the out-proxy place and the out-proxy place sends the token to its corresponding in-proxy place. To send the token to the corresponding in-proxy place, the agent refers to the team organization to figure out the agent to which the role associated with the action corresponding to T2 is delegated.
- The second is to guarantee a collaborative action among roles, a plan invocation, goal achievement, or role variable selection. This situation appears as each of the agents to which the involved roles are delegated has a corresponding special transition in its individual process. The firing of such a special transition requires coordination with other corresponding transitions. As above, each individual process refers to the team organization to figure out the agents to which the roles associated with the action are delegated so that coordination takes place among corresponding transitions.

With the execution of the individual processes, the team organization is updated and further drives dynamic composition of individual process as described below:

- If a team of agents starts a plan invocation or achieving a goal, a corresponding node is added into team organization as a child of its root, and the involved agents add RoB-CAST-PN components into their individual processes. Later, the involved agents remove the RoB-CAST-PN components from their individual processes once the plan invocation is completed or the goal is achieved. This procedure is described in Sec. V.5.3.1.

- If a goal transition is fired and its coordination decides to use a role-based plan to achieve its goal, its corresponding node in the team organization records the role-based plan, and the involved agents dynamically replace their corresponding goal transitions with the corresponding plan transition in their individual processes. This is realized as described in Sec. V.5.3.1, i.e., the goal transition is expanded by a net with a plan transition only (the firing of a plan transition is discussed in next item);
- If a plan transition is fired and its coordination determines a team assignment for the plan invocation, its corresponding node in the team organization records the team assignment in its team structure, and the involved agents translate the responsibility of the role(s) delegated to them into a RoB-CAST-PN as described in Sec. V.5.3.2. Then the corresponding plan transition is dynamically expanded with the RoB-CAST-PN as described in Sec. V.5.3.3. Later, the involved agents shrink the RoB-CAST-PN corresponding to the plan invocation back to the corresponding plan transition in their individual processes once the plan invocation is completed, as described in Sec. V.5.3.3;
- If a selection transition is fired and its coordination decides which role is selected, its corresponding node in team organization records the role variable selection in its team structure, the agent to which the previously selected role is delegated removes the RoB-CAST-PN corresponding to the responsibility of the role variable from its individual process, and the agent to which the currently selected role is delegated translates the responsibility of the role variable into a RoB-CAST-PN as described in Sec. V.5.3.2 and then appends the RoB-CAST-PN into its individual process as described in Sec. V.5.3.3.

In the above interaction, an agent maintains its copy of team organization according to its mode, relevant or full tracking.

V.5.5 Execution of Individual Process

As explained earlier, the individual process of an agent contains the portion of a team process related to it. An agent executes the team process by executing its individual process represented by a RoB-CAST-PN. As explained in Sec. V.5.2, we have introduced proxy places to help represent temporal ordering on actions represented by transitions in different individual process; we have introduced special transitions to handle the coordination between agents for complex actions. With these extensions, the agent can just check the enabled transitions locally and then fires the enabled transitions. In this way, an individual process can be executed as a traditional Petri net is executed in a single platform even though the individual process is part of a team process.

Suppose there is an individual process represented by a RoB-CAST-PN, N = (P, T, C, D, ϕ, F, G, L, M_0, T_e, T_x, T_s, N_s, S). The individual process is executed by firing transitions. A transition fires by removing tokens from its input places and creating new tokens that are distributed to its output places. A transition may fire if it is enabled. A transition t in N with M is enabled if:

1. for all $p \in {}^\bullet t$, $M(p) \geq L(p, t)$; and
2. the condition c in $\phi(t) = (c, d)$ is true within the duration d since (1) is satisfied.

A transition t in a marked RoB-CAST-PN N with marking M may fire whenever it is enabled. Firing an enabled transition t results in a new marking M' defined by

$$M'(p) = M(p) - L(p, t) + L(t, p) \times gv(t, p)$$

Given any two enabled transitions t_1 and t_2 under marking M, they are concurrent firings if

$$\forall\, p \in {}^\bullet t_1 \cap {}^\bullet t_2,\ M(p) \geq L(p, t_1) + L(p, t_2)$$

To maximize the concurrency of the execution of the individual processes the agents in teamwork, the RoB-CAST-PNs of the individual processes are executed asynchronously. The asynchronous execution of a RoB-CAST-PN is realized by a firing mode, called Fire and Forget (FF). The Fire and Forget mode is that, an agent keeps searching for enabled transitions; if a transition is enabled, the transition fires and tokens in its input places are removed according to the above firing rule; the agent keeps

searching for enabled transitions[17]; and if the firing of the transition is done, tokens are put in its output places according to the above firing rule. The Fire and Forget mode can be implemented by using a separate thread for firing an enabled transition.

Besides changing the markings of a RoB-CAST-PN, firing a special transition drives the agent to perform additional processing. If an agent fires a special transition *t*, then the agent calls the corresponding routine describe in V.5.2.2.2 to perform the corresponding processing (particularly, the agent dynamically composes its individual process as described in Sec. V.5.3.3 if *t* is a plan transition); moreover, the agent dynamically updates its team organization as described in Sec. V.5.4.

For easier description, we use Fire(*IP*, *t*) represent the firing of transition *t* in *IP*. Fire(*IP*, *t*) is a separate thread which performs additional processing if *t* is a special transition and put tokens to its output places. Also, if a transition, say *t1*, has a same input place as *t* which contains not enough tokens to fire *t1* after firing *t*, the transition *t1* is in conflict with *t*. The following is the algorithm for executing individual process *IP*.

Algorithm 16. Executing an individual process *IP*

```
Execute(IP)
  Begin
    Create an empty list, Enableds;
    Loop
      Search all enabled transisitons in IP and save them in list Enableds;
      While Enableds  is not empty do
        Randomly select a transition t from Enableds;
        Remove t from Enableds;
        Fire(IP, t);    /* Recall this call just starts a separate thread.*/
        Remove all transitions in conflict with t from Enableds ;
      End-While;
```

[17] Firing an operator takes time because the operator may be with duration. According to the firing rule of RoB-CAST-PN, a transition can fire no matter whether other transitions are firing. So, the execution of a RoB-CAST-PN keeps searching enabled transitions when some transitions are firing.

Remove all places, transitions and arcs for the invocation corresponding to the end place if any end place in IP has a token;

End-Loop;

End

In Chapters III and IV, we have stated that termination conditions can be specified to monitor a plan execution. To monitor the execution of a plan in shared mental model, a routine[18] can be added in the loop of the algorithm to monitor its termination conditions. Once the conditions are satisfied, the agent notifies all involved agents. The agents reduce the RoB-CAST-PN components corresponding to the plan execution to the corresponding plan transition.

V.6 Reasoning Based on Shared Mental Models

Psychological studies on human teamwork have shown that the members of an effective team [65, 85, 87] often maintain mutual awareness, coordinate to resolve conflicts, anticipate the needs of teammates, predict likely changes, provide help to incapable teammates, and backup the failures of disabled teammates. Shared mental models are the fundamental of appropriate reasoning mechanisms of forming expectations and explanations which foster those favorite features of effective teams [85].

Our role-based shared mental models support various reasoning methods to achieve effective teamwork. In this section, we will introduce two reasoning mechanisms, role-based proactive information exchange and proactive helping behavior, to improve teamwork effectiveness. The representation of a team process in our role-based mental model accommodates the mechanism of task allocation by delegating roles. The agents only maintain a low level of overlapping (i.e., team organization) and each agent maintains the portion of team process related to itself as individual process.

[18] In RoB-CAST, this routine has been implemented and embedded at the end of the loop in Algorithm 16. It checks the termination conditions of ongoing plan invocations. If the termination conditions of a plan invocation become true, the agent synchronizes all other involved agents to stop the execution of the plan invocation, including all its sub-plan invocations. To avoid diverging our focus on the description of the execution of individual processes, we leave the detail about the routine out.

Consequently, their teamwork is improved by less communication among the agents and faster execution of individual processes. These two mechanisms are mainly based on such representation of team process with a low degree of overlapping.

V.6.1 Mutual Awareness

Shared mental models facilitate mutual awareness among team members [65, 85, 87], i.e., an individual can determine not only its situation but also those of its team members. In this section, we will introduce how mutual awareness can be realized based on our role-based shared mental models which are with a low level of overlapping, team organization. We will discuss the trade-off between the level of overlapping and the accuracy of mutual awareness. A few mechanisms of reasoning about the rough time of action performance by other agents are also explained.

In the RoB-SMMs, team organizations can make agents mutually aware of what other agents do. We assume that every agent knows all operators supported by the domain, including their preconditions and effects, and all role-based plans, including the roles and role variables in the plans and actions associated with them. A team organization maintains the goals in which agents are involved, the role-based plans by which the agents achieve the goals, and the team structure to perform the plans. Based on this information, an agent can infer what another agent's actions are. Given a role-based plan, an agent can decompose a team process into individual responsibilities and further infer what an individual is expected to do. If agent *ag′* believes that a role *r* in a role-based plan *P* is delegated to an agent *ag*, then agent ag' can infer that agent *ag* intends to execute the actions associated with the role. Moreover, if action *a* is associated with role r (we use **Do**(*r*, *a*) to represent that action *a* is associated with role r), then then agent ag' can infer that agent *ag* intends to execute action *a* (we use **Intend**(*ag*, *a*) to represent that agent *ag* intends to execute action *a*). This can be characterized by

Bel(*ag′*, **Assign**(r, ag)) $\rightarrow$ (**Bel**(*ag′*, **Resp**(r, P)) $\wedge$ **Bel**(*ag′*, **Intend**(*ag, r*, *P*)))

(**Bel**(*ag′*, **Intend**(*ag*, *r*, *P*)) ∧ **Bel**(*ag′*, (**Do**(*r*, *a*) ∈ **Resp**(*r*, *P*))))→ **Bel**(*ag′*, **Intend**(*ag*, *a*))

Moreover, every agent knows other agents have the ability to do the same reasoning. Thus every agent can further believe that all other agents have the same beleiefs on agents' actions, that is, agents have mutual awareness about agents' actions.

Team organizations enable agents to be mutually aware of each other even though the agents are involved multiple simultaneous tasks. When agents dynamically invoke sub-plans, the agents dynamically compose their individual processes and update their team organizations correspondingly. Agents can maintain team organization by different modes. Full tracking enables an agent to keep aware of what roles and/or role variables other agents are doing in a plan invocation even though the agent does not participate in the invocation. In previous work of our MURI group, the shared mental model allowed the involved agents to maintain their team process represented by CAST-PNs (each agent has a copy of CAST-PN, and their copies might not be consistent) and the CAST-PNs are constructed before executing their team process. This requires that all plan invocations are included in a top-level plan before executing the top-level plan. However a team often contains a couple of sub-teams which are involved in multiple tasks at the same time. The sub-teams may be formed to invoke a sub-plan on the fly. So, CAST-PNs cannot handle multiple dynamic and simultaneous tasks, not even to say the mutual awareness across multiple dynamic and simultaneous tasks.

Moreover, mutual awareness via team organization does not require maintaining a consistent team process[19]. Maintaining a consistent team process[20] in CAST 3.0 caused a great amount of communication and further compromises the effectiveness of teamwork.

---

[19] A consistent team process occurs when every agent's knowledge of the team process containing the execution status is the same. Each involved agent has a copy of the knowledge, including the execution status of both itself and other agents. It is called consistent when the agent must update the change to other agents so that they can reflect the change in their copies once an agent changes its execution status.

[20] In CAST 3.0, although agents maintain a copy of (paritally) consistent team process (i.e., CAST-PN), agents may have inconsistent beliefs that lead to inconsistent executions. For example, agent ag1 believes φ while agent ag2 believes $\neg\varphi$. In RoB-CAST, agents may also have inconsistent beliefs. However, RoB-MALLET has clear semantics on whether the evaluation of a condition φ is based on mutual or individual belief about φ. Thus, inconsistent beliefs would not cause inconsistent executions in RoB-CAST. This has been discussed in Chapter III.

In many dynamic domains, such as military battlefields, it is not realistic that individuals have a complete team process. This prevents individuals from maintaining a consistent and detailed team process. On the other hand, maintaining a low level of overlapping, team organization, is compatible with such reality. An individual cannot know the team process about others, but he/she may knows what task assignments others have or what roles others play.

As a tradeoff for maintaining a low level of overlapping, a team organization cannot enable an agent to determine exactly when another agent performs an action. In a role-based plan, each role is associated with a bag of actions. If a role is delegated to an agent *ag1*, other agents can infer that the agent performs the actions associated with the role. Another agent *ag2* cannot derive the exact time when the agent *ag1* is performing the actions, however, the rough execution time may, in some circumstances, be derived based on the temporal orders or coordination between the actions associated with the roles delegated to *ag1* and *ag2*. Suppose that ag_1 performs a_1 and ag_2 performs a_2.

- If there is a temporal order (a_1, a_2) (we represent it by **Order**(a_1, a_2)), ag_1 and ag_2 can derive the relative order of the performance of a_1 and a_2. This can be characterized by:

  $$\textbf{Bel}(ag, \textbf{Order}(a_1, a_2) \wedge \textbf{Executed}(ag_1, a_1)\ ) \rightarrow \textbf{Bel}(ag, \textbf{Execute}\ (ag_2, a_2))$$[21]

  $$\textbf{Bel}(ag, \textbf{Order}(a_1, a_2) \wedge \textbf{Executing}(ag_2, a_2)\ ) \rightarrow \textbf{Bel}(ag, \textbf{Executed}\ (ag_1, a_1))$$

- If there is an AND coordination between a_1 and a_2 (specified by Joint Do in MALLET. We represent it by **AND**(a_1, a_2)), agent ag can infer that agents ag_1 and ag_2 execute a_1 and a_2 respectively at the same time. This can be characterized by:

  $$\textbf{Bel}(ag, \textbf{AND}(a_1, a_2) \wedge \textbf{Executing}(ag_1, a_1)\ ) \rightarrow \textbf{Bel}(ag, \textbf{Executing}\ (ag_2, a_2))$$

  $$\textbf{Bel}(ag, \textbf{AND}(a_1, a_2) \wedge \textbf{Executing}(ag_2, a_2)\ ) \rightarrow \textbf{Bel}(ag, \textbf{Executing}\ (ag_1, a_1))$$

---

[21] We use **Execute**(*ag*, *a*), **Executing**(*ag*, *a*), and **Executed**(*ag*, *a*) to respectively represent that action *a* will be executed next, is being executed now, and has been executed by agent *ag*. We use **Execute**(*T*, *a*), **Executing**(*T*, *a*), and **Executed**(*T*, *a*) to respectively represent that action *a* will be executed next, is being executed now, and has been executed by the agents in team *T*. Note that the notion of "next" does not imply any global ordering among teams or individuals not part of T; other individuals or teams could be doing things at the same time (on other machines) as *T*.

- In the environment where individuals have some degree of observability, an agent may perceive the performance of some actions by other agents. Then, according to the formalisms in the above two items, the agent can derive the rough execution time of other agents according to temporal orders between those actions and other actions in a team process.
- Some actions may be conditional. Probabilistic mechanisms, such as Bayesian reasoning [115], can be applied the time of their possible occurrences.

If necessary, additional communication might be required to query the execution status of other agents. In Sec. V.6.3, we use additional communication to check the execution status of other agents to decide whether they really need help. Even so, the total communication is still much lower than that for maintaining consistent team process. One may argue fully consistent team process is not necessary. If team process is only partially consistent, the agents will have the same problem: cannot decide the exact timing of the performance of actions by other agents.

V.6.2 Role-Based Proactive Information Exchange

V.6.2.1 Introduction

Shared mental models facilitate information sharing among team members. Stasser and Titus [95] suggested that groups are able to make better decisions by pooling information. Mohammed and Dumville [71] categorized the research on this topic in different fields into information sharing, transactive memory, group learning, and cognitive consensus, and examine their disciplinary boundaries. Previous works [124, 125, 126, 128] in our MURI group simulated effective teamwork by proactive information exchange, which functions on shared mental models, particularly team process.

Proactive information exchange is one of the important features distinguishing CAST from other teamwork architectures. Our RoB-CAST preserves this attractive feature by a mechanism of role-based proactive information exchange, which extends proactive information exchange to role-based shared mental models. The main

difference between our role-based information exchange and previous proactive information exchange is, our role-based proactive information exchange is based on mutual awareness explained in last section while previous information exchange is directly based on consistent team process. As a result, role-based proactive information exchange has two advantages over previous proactive information exchange:

- Both mechanisms are based on shared mental models and have equivalent functionality; however, our role-based shared mental models only maintain a low level of overlapping (i.e., team organizations), instead of the entire team process in previous shared mental models, and thus require much less communication. Therefore, the cost in our role-based proactive information exchange is much lower than that in previous proactive information exchange.
- Our role-based proactive information exchange is more flexible. Previous proactive information exchange requires agents to share team processes, so it only functions among the agents involved in the same plan invocation. Our information exchange only requires mutual awareness. It even functions among the agents involving different simultaneous plan invocations. As pointed out earlier, an individual are limited to have the detail about the process knowledge of its teammates.

V.6.2.2 Formalism of Role-Based Proactive Information Exchange

The reasoning mechanism for role-based proactive information is formalized by two modal operators, **Inform**(ag_1, ag_2, I) and **Ask**(ag_1, ag_2, I). **Inform**(ag_1, ag_2, I) means that agent ag_1 is obligated to inform agent ag_2 of I. **Ask**(ag_1, ag_2, I) means that agent ag_1 is obligated to ask agent ag_2 of I. We formalize the conditions of **Inform**(ag_1, ag_2, I) and **Ask**(ag_1, ag_2, I) as follows:

$$\begin{aligned}
&\mathbf{Bel}(ag_1, I)\\
&\wedge\ \mathbf{Bel}(ag_1, \neg\, \mathbf{Bel}(ag_2, I))\\
&\wedge\ \mathbf{Bel}(ag_1, \mathbf{Intend}(ag_2, r, P)) \wedge\ \mathbf{Bel}(ag_1, \mathbf{Do}(r, a))\\
&\wedge\ \ \mathbf{Bel}(ag_1, \mathbf{Precond}(I, a))
\end{aligned}$$

$\rightarrow$ **Inform**(ag_1, ag_2, I)

$\neg$ **Bel**(ag_1, I)
$\wedge$ **Bel**(ag_1, **Bel**(ag_2, I))
$\wedge$ **Bel**(ag_1, **Intend**(ag_1, r, P)) $\wedge$ **Bel**(ag_1, **Do**(r, a))
$\wedge$ **Bel**(ag_1, **Precond**(I, a))
$\rightarrow$ **Ask**(ag_1, ag_2, I)

where, modal operator **Bel**(ag, I) means that ag believes I; modal operator **Precond**(I, a) means that I is a precondition of action a (i.e., if I is not true, then action a cannot be done, however, if I is true, action a can be done unless action a is terminated by termination conditons).

These two modal operators characterize the conditions under which information exchange takes place. Agent ag_1 sends information I to agent ag_2 when:

- ag_1 believes I;
- ag_1 believes that ag_2 does not believe I;
- ag_1 believes that ag_2 intends to take the responsibility of role (or role variable) r in plan P and action a is associated with r;
- *ag1* believes that I is a precondition of action a.

Agent ag_1 asks agent ag_2 information I when:

- ag_1 does not believes I;
- ag_1 believes that ag_2 believes I;
- ag_1 believes that it intends to take the responsibility of role (or role variable) r in plan P and action a is associated with r;
- *ag1* believes that I is a precondition of action a.

In these two modal operators, an agent needs to know the need for information I (i.e., condition 4 in the above formulas) and whether its peer believes I (i.e., condition 2 in the above formulas) to determine whether information exchange is really necessary. The need for information I by an agent can be determined by the preconditions of operators which the agent is to perform. Whether the peer of an agent believes I can be inferred

from the effects of actions that have been done. In Chapter III, we have explained that the performance of operators is a way to change agents' beliefs. Once an operator is completed by an agent, the agent's knowledge base, which maintains the agent's beliefs, should reflect the state transition from its preconditions to its effects. By using mutual awareness, an agent can infer what actions another agent has done and further its beliefs. This inference can be characterized by the following formula:

$$\mathbf{Bel}(ag_1, \mathbf{Intend}(ag_2, r, P) \wedge \mathbf{Executed}(ag_2, a)) \wedge (\mathbf{Executed}(ag_2, a) \rightarrow \mathbf{Bel}(ag_2, I)) \rightarrow \mathbf{Bel}(ag_1, \mathbf{Bel}(ag_2, I))$$

One may question, now that an agent ag_1 believes that another agent ag_2 believes information I, why agent ag_1 still needs to ask agent ag_2 for I, instead of directly believe\ing I? We note that, there is some approximation on inferring other agents' beliefs through the actions done by other agents. As described in earlier chapters, actions may be parameterized by variables and the effects of actions may include the variables. An agent may not know the exact values for the variables. Consequently, the information I inferred by the agent may contain variables. Also, some actions associated with roles could be conditional (actions in IF and WHILE statements in MALLET). For such a conditional action a, mutual awareness based on team organization cannot exactly determine whether a would be done. For these reasons, agent ag_1 still needs to ask ag_2 for I, more precisely, the bindings of I.

We note that, the condition (2) of both of **Inform**(ag_1, ag_2, I) and **Ask**(ag_1, ag_2, I) requires that agent ag_1 have beliefs on agent ag_2's beliefs. Agents in RoB-CAST only infer other agents' beliefs through inferring the actions done by other agents. They cannot predict other agents' beliefs updated by other causes, such as observability [131]. Ioerger's PIEX [42] considers several types of causes of updating beliefs. An agent proactively sends a message about some information only if the agent can infer that the other agent either does not believe the information or believes it incorrectly. As a result, un-necessary information exchange can be reduced. We believe that PIEX can be implemented together with our Role-Based Proactive Information exchange. Role-based

proactive information exchange might also be refined by accommodating decision-making theory [91] to reason about whether other agents perform conditional actions.

V.6.2.3 Algorithms of Role-Based Proactive Information Exchange

To distinguish from the algorithms of previous proactive information exchange, called DIARG (Dynamic Inter-Agent Rule Generator) [128], we call the algorithms of role-based proactive information exchange RoB-DIARG (Role-Based DIARG). Similar to DIARG, RoB-DIARG consists of offline and online algorithms. However, the offline and online algorithms in RoB-DIARG are different from those in DIARG. The offline algorithm in DIARG generates all possible information flows represented by 3-tuple (Predicate, Providers, Needers). The online algorithm in DIARG filters possible information flow according to team process. The offline algorithm in RoB-DIARG generates all possible information providers and needers, which are roles and role variables in role-based plans. Information providers and needers are represented by 3-tuple (Predicate, Plan, Providers) and 3-tuple (Predicate, Plan, Needers) respectively. The online algorithm infers all possible information provider agents and needer agents according to team organization. By matching the predicates of providers and needers, the online algorithms dynamically decide information flows.

The offline algorithm in RoB-DIARG functions on the basis of the generation of team responsibility (described in Sec. IV.4.3), the partition of team responsibility into individual responsibilities (described in Sec. IV.5) and the translation of individual responsibility into RoB-CAST-PN representation (described in Sec. V.5.3.2).

Algorithm 17. The offline algorithm of RoB-DIARG for a role-based plan P

RoB_DIARG_Offline(*P*)

Begin

Create information providers list *InfoProviders*;

Create information needers list *InfoNeeders*;

TeamResp = GenerateTeamResponsibility (*P*);

```
Let Start be the start node of TeamResp;
Let End be the end node of TeamResp;
PartitionTeamResponsibility (TeamResp, Start, End);

For each role or role variable r in P do
   Let Resp be the individual responsibility of r;
   PN = Translate_Responsibility(Resp);
   For each transition t in PN do
      If t is operator transition
         Let precond be the precondition of the operator corresponding to t;
         Let effect be the effect of the operator corresponding to t;
         Add (precond, P, r) into InfoNeeders;
         Add (effect, P, r) into InfoProviders;
      ElseIf t is plan transition
         Let P' be the plan corresponding to t;
         RoB_DIARG_Offline(P');
         For each role r' in P' possibly assigned to r do
            For each (pred, P', r') in InfoNeeders of RoB_DIARG_Offline(P') do
               Add (pred, P, r) into InfoNeeders;
            End-For;
            For each (pred, P', r') in InfoProviders of RoB_DIARG_Offline(P') do
               Add (pred, P, r) into InfoProviders;
            End-For;
         End-For;
      End-If
   End-For;
End-For;
End;
```

The above algorithm recursively calls itself. If P calls directly or indirectly itself (i.e., *P'* in some hierarchical level is *P*), special processing is required to avoid infinite recursion. It can be done by remembering this occurrence and finding its closure later.

The online algorithm of RoB-DIARG implements modal operators Inform and Ask. After an agent has new information, either by observation or by completing the performance of an operator, it triggers the Inform routine to send the information to other agents who need the information. If an agent needs some information that it does not believe, it triggers the Ask routine to query this information from other agents, which possibly believe the information. We assume agents have certain communication facility. In the online algorithm of RoB-DIARG, we use Send(*ag*, *I*) to represent the communication of sending information *I* to *ag* and Ask(*ag*, *I*) to represent the communication of asking information *I* from *ag*.

Algorithm 18. The online algorithm of RoB-DIARG

```
Inform()
  Begin
    If I is newly sensed information
      Check team organization and find out all plan invocations;
      For each plan invocation P do
        Check information needers Needers generated by RoB_DIARG_Offline(P);
        For each information needer (Pred, P, r) do
          If I matches Pred
            Check team organization and find out which agent plays r;
            Let ag be the agent plays r;
            Send(ag, I);
          End-If;
        End-For;
      End-For;
    End-If;
  End;
```

```
Ask()
  Begin
    If I is needed
      Check team organization and find out all plan invocations;
      For each plan invocation P do
        Check information providers Providers generated by
RoB_DIARG_Offline(P);
        For each information provider (Pred, P, r) do
          If I matches Pred
            Check team organization and find out which agent plays r;
            Let ag be the agent plays r;
            Ask(ag, I);
          End-If;
        End-For;
      End-For;
    End-If;
  End;
```

We must point out that the mechanism of dynamic role selection in previous proactive information exchange [127, 128] is totally different from our task delegation described in Chapter IV. Its concept of role actually is an alternative way to specify capability requirement for an agent to play a role (i.e., a set of operators). It is not associated with actions. To some extend, it is similar to the concept of position used in our teamwork architecture. It cannot facilitate various advantages originated by our concept of roles in our teamwork architecture, including task decomposition and delegation, efficient representation of shared mental models, and further mutual awareness with a low level of overlapping.

V.6.3 Role-Based Proactive Helping Behavior

V.6.3.1 Introduction

Brehm and Kassin [9] identified the motivations of helping behavior from biological factors (i.e., a creature has a tendency, originated from natural selection, to reciprocal helping), emotional factors (i.e., a person's empathy makes him/her help to reduce the distress of another person) and social normative factors (i.e., social norms promote help giving in social contexts). Lind [63] proposed a dual-aspect-theory of moral development and helping behavior to distinguish a person's desire to help and his/her ability to help adequately and further hypothesized the conditions of triggering helping behavior. The dual-aspect-theory of helping behavior can be viewed as a hypothesis about how social norms actually motivate helping behavior.

Miceli, Cesta and Rizzo [67] described the conditions and motivations for seeking and giving help based on social dependence between two agents. They defined social dependence of agent x on agent y as a situation in which x has a goal φ, x is unable to do action a, y is able to do action a, and a is an action to achieve φ. They demonstrated that "social" agents with the attitudes of help seeking and help giving are the most successful, comparing with "solitary" agents without such attitudes [67]. They also verified that such social strategy of help giving and help seeking is robust in several risky conditions [17].

Backing up behaviors are a special kind of helping behaviors in the situation where some team member(s) fails to accomplish a certain action and other team member(s) take an action to cover what the failure action targets. Porter et al. [79] proposed a Five Factor Model of personality to describe the key characteristics of backing up behaviors, including back up recipients, back up providers, and the legitimacy of the needs for backing up.

Our role-based shared mental models can enable the agents in a team to provide proactive helping behaviors to each other. While many social morals, such as laws, religions and cultures, might affect helping behavior, we focus on how to recognize the needs of helping and how to provide helping behaviors correspondingly.

Our reasoning mechanism can facilitate proactive helping behaviors between teams rather than only between two agents. An agent recognizes the need for help of a team through mutual awareness based on the role-based shared mental. If an agent finds out a course of actions by which a team of agents can cover the helping need, it coordinates with those agents to perform the actions. The notion of social dependence [67] based on two agents' capabilities of some action can build up the relationships of help giving and help seeking and further facilitate helping behavior between two agents. However, if a team of agents is to perform a course of actions, every action must be capable by some agent(s) and there must be an admissible delegation of the actions to the agents. Our reasoning mechanism of role-based proactive helping behavior ensures the coverage of capability requirements and the admissible delegation of the actions to the agents.

Helping behaviors may take place in two ways: help others to finish what they are doing and take over what others are doing if they fail (i.e., back up). Our reasoning mechanism of role-based proactive helping behavior identifies these two types of help needs.

V.6.3.2 Formalism of Role-Based Proactive Helping Behavior

The reasoning mechanism of role-based proactive helping behavior can be defined by a meta-predicate **Help**(G_1, G_2, φ', φ, a) which characterizes the team which needs help, the team which provides helping behavior, the conditions under which a proactive helping behavior takes places, and what action is taken. If **Help**(G_1, G_2, φ', φ, a), then G_1 executes action a to help G_2 to achieve its joint persistent goal φ.

The formalism enables two types of conditions of helping behavior: backup behaviors and promotion behaviors (i.e., reaching the prerequisite of goal φ). For easier description, we explain **Help**(G_1, G_2, φ', φ, a) with these two types of conditions separately. For backup behaviors, team G_1 proactively helps G_2 for goal by action a when:

1. G_1 and G_2 mutually believes that G_1 and G_2 have a joint persistent goal φ', that G_2 has a joint persistent goal φ, and that φ is a sub-goal of φ';
2. G_1 mutually believes that G_2 will believe φ if G_2 has done action a;

3. G_1 mutually believes that G_2 will never have doneaction a ;
4. G_1 mutually believes that φ will be true if G_1 performs action a;
5. G_1 is capable of action a.

For promotion behaviors, team G_1 proactively helps G_2 for goal by action a when:

1. G_1 and G_2 mutually believes that G_1 and G_2 has a joint persistent goal φ', that G_2 has a joint persistent goal φ, and that φ is a sub-goal of φ';
2. G_1 mutually believes that G_2 will believe φ if G_2 has done action a
3. G_1 mutually believes that there exist ψ which is not true and that no action is being executed by any other team to make ψ true;
4. G_1 mutually believes that ψ is a precondition of action a;
5. G_1 mutually believes that ψ will be true if G_1 performs action a;
6. G_1 is capable of action a.

In our teamwork architecture, an action taken by agents could be either an operator or a role-based plan. We assume every agent knows the knowledge about all operators and role-based plans, including their preconditions, their effects, and termination conditions for role-based plans. Such knowledge is part of shared mental model together with team process as described in Sec. V.2. Agents achieve goals by performing some of the actions, initially by invoking role-based plans. Through role-based shared mental models, agents can mutually be aware of what other agents are doing, including what goals other agents are achieving, what role-based plans are used for the goals, team structure of performing the role-based plans, and a coarse knowledge about the individual processes of the agents involved in the role-based plans (i.e., the agents know their RoB-CAST-PN representation of individual processes but not concrete markings). We note that mutual beliefs among can be determined. In Sec. III.3.2, we have explained how a mutual belief on a condition is determined. In Sec. V.6.1, we have explained how mutual awareness is enabled (i.e, agents can infer an agent *ag*'s status; and, moreover, agents believe that other agents can infer agent *ag*'s status). Based on such knowledge, agents can identify the conditions in the above formula as the following:

- condition 1 in both types of helping behaviors can be identified through team organization and individual process;
- condition 2 in both types of helping behaviors can be identified based on the specifications of operators and role-based plans, specifically the effects of operators/plans;
- condition 3 in the backup behaviors is the failure of achieving goal φ. It can be recognized through termination conditions which are treated as the cause of failures. One may argue that only plans have termination conditions while operators are atomic and thus have no termination conditions. Considering the main focus of our teamwork architecture is to utilize role-based plans and operators can be wrapped by role-based plans with termination conditions, our formalism still does not lose its generality;
- conditions 3 and 4 in promotion behaviors are to recognize the lack of prerequisite ψ of achieving goal φ. In other words, agents neither believe ψ nor do any action to make ψ true. Agents can know what action can be taken to achieve goal φ, so they can identify such lack by checking the preconditions of the action. In MALLET, there are three types of precondition, fail, wait and achieve. The details about these types of preconditions are available in Chapter III. In the algorithm of proactive helping behavior, false wait preconditions are treated as the clue to identify the lack of prerequisite of achieving a goal. Agents can infer if any current action turns ψ true by comparing the effects of the action with ψ. It is possible that some current action cover the prerequisite ψ of achieving goal φ. If so, no helping behavior needs to be initialized;
- condition 4 in backup behaviors and condition 4 in the second type are to identify if action a is proper to realize goal φ or cover the prerequisite ψ of achieving goal φ. This can be done by comparing the effects of action a with φ or ψ correspondingly;
- condition 5 in backup behaviors and condition 6 in promotion behaviors are to identify which agents are suitable to perform action a. Action a may be a course

of operators and role-based plans. To decide if a team is suitable for the course, every operator should be capable by some agent(s) and there is an admissible team assignment for every role-based plan. The mechanism of searching admissible team assignment has been explained in Sec. IV.6.

In fact, the determination of a proper action and the determination of a proper team to provide helping behaviors interact with each other. If the effects of an action can cover the goal but there is no admissible team assignment to perform the action, the action is not proper and the searching of a proper action continues to examine other actions. The mechanism of planning proper actions has been discussed in IV.7.

It is possible that multiple actions with admissible team assignment are available. Other reasoning mechanisms can be accommodated to decide the best action and team assignment. For example, the best action should cost as low resource as possible and team assignment should balance the workload of team members.

We admit that our formalism only captures the failure of an action and then finds back up. In many situations, the failure of action may be just caused by an agent failing to perform some recipe of the action. In the invocation of a role-based plan, such failure appears as an agent fails to perform the responsibility of the role delegated to the agent. To cover such failure, the role can be delegated to another agent to resume the rest of responsibility of the role and the agents to which other roles are delegated remain their performance of the responsibilities of those roles.

V.6.3.3 Algorithms of Role-Based Proactive Helping Behavior

The algorithm of role-based proactive helping behavior consists of offline and online algorithms. The offline algorithm generates all potential helping needs (i.e. the wait preconditions of actions) and all coverage of possible helping needs (i.e., the effects of actions). The online algorithm filters the potential needs of would-do actions with the coverage of possible helping needs by on-going actions and decides a proper action for a proper team to provide helping behavior. The offline algorithm is run on role-based plans statically before the plans can be utilized for helping behavior. The online algorithm is run by each agent in a separate thread all the time.

The offline algorithm functions on the basis of the generation of team responsibility (described in Sec. IV.4.3), the partition of team responsibility into individual responsibilities (described in Sec. IV.5) and the translation of individual responsibility into RoB-CAST-PN representation (described in Sec. V.5.3.2). A potential helping need is represented by a 4-tuple (Predicate, Plan, Needers, Transition). A coverage of possible helping need is represented by a 2-tuple (Predicate, Plan). The offline algorithm is statically run on a role-based plan instead of its invocation, so the transition in the 4-tuple is named without being prefixed with concrete plan invocation as the naming convention introduced in Sec. V.5.3. To identify the actual transition in the individual process, it must be prefixed by plan invocation.

Algorithm 19. The offline algorithm of role-based proactive helping behavior for a role-based plan P

```
RoB_Helping_Offline(P)
  Begin
    Create helping need list HelpNeeds;
    Create coverage list HelpEffects;

    TeamResp = GenerateTeamResponsibility (P);
    Let Start be the start node of TeamResp;
    Let End be the end node of TeamResp;
    PartitionTeamResponsibility (TeamResp, Start, End);

    For each role or role variable r in P do
      Let Resp be the individual responsibility of r;
      PN = Translate_Responsibility(Resp);
      For each transition t in PN do
        Let precond be the precondition corresponding to t;
        If precond is a wait condition
          Add (precond, P, r, t) into HelpNeeds;
```

End-If

Let *effect* be the effect of the operator corresponding to *t*;

Add (*effect*, *P*) into *HelpEffects*;

End-For;

End-For;

End;

The online algorithm filters the potential needs of would-do actions with the coverage of possible helping needs by on-going actions and decides a proper action for a proper team to provide helping behavior.

The online algorithm for role-based proactive helping behavior refers to the team organization and finds out the current plan invocations. Through the offline algorithm, it can infer all potential helping needs and all coverage for potential helping needs of the invocations. However, it is not necessary that all potential help needs will be actual help needs. Agents may participate in multiple tasks. Some potential help needs may be realized by actions in other simultaneous tasks. Therefore, those implied by the effects of the current actions can be reduced. As long as mutual awareness based on the role-based shared mental models does not have exact tracking of the actual execution of other agents' actions, extra communication is required to check if helping is actually needed and. Moreover, if helping is actually needed, then the agents, which wish to provide helping behaviors, need to communicate with the agents, which need helping to find out the exact help needs. The reason is, the predicates of potential help needs extracted by the offline algorithm may contain variables. The agents that need helps dynamically bind the actual values to these variables and the values are unknown to the agents which wish to provide helping behaviors. So communication must take place to make the agents which wish to provide helping behaviors know the bindings. . The online algorithm utilizes a role-based plan to provide helping behavior even though complex planning mechanisms might be accommodated. In Sec. IV.7, we have explained how role-based plans can be used in complex planning algorithms and presented a simple version of a planning algorithm to search a role-based plan for a goal, denoted by Achieve(*AT, Goal,*

PlanLibrary). After a proper role-based plan is found, the team to perform the plan can be decided as the subset of all available agents which can make an admissible team assignment for the plan invocation. The algorithm of searching for an admissible team assignment, denoted by SearchTeamAssignment (*AT, P*), has been introduced in Sec. IV.6.3.

Algorithm 20. The online algorithm of role-based proactive helping behavior

```
RoB_Helping_Online()
  Begin
    Let AT be the team of all agents;
    Let PlanLibrary be the library of all role-based plan;
    Loop
      For each plan invocation Inv in the team organization do
        Let P be the plan of Inv;
        Let HelpNeeds be the helping need list generated by
RoB_Helping_Offline(P);
        For each helping need Need in HelpNeeds do
          Bindings = Actual_Help_Needed(Inv, Need);
          If Bindings ≠ null
            Let HelpGoal be the predicate in Need with Bindings;
            Plan = Achieve(AT, HelpGoal, PlanLibrary);
            If Plan ≠ failure
              Assignment = SearchTeamAssignment (AT, P);
              Let G be the agents actually in Assignment;
              Notify the agents in AT that G provides helping for Need in Inv;
              Ask G start Plan;
            End-If;
          End-If;
        End-For;
      End-For;
```

```
    End-Loop;
  End;

Actual_Help_Needed (Inv, Need)
  Begin
    (pred, Plan, G, t) = Need;
    If Need in Inv has been helped
      Return null;
    Else
      Create a list NoNeeds;
      For each plan invocation Inv in team organization do
        Let P be the plan of Inv;
        Let HelpEffects be the coverage list generated by RoB_Helping_Offline(P);
        Add all predicates in HelpEffects to NoNeeds;
      End-For;
      If the predicates in NoNeeds implies pred
        Return null;
      Else
        Check team organization to find the agents (T) delegated G;
        Ask one agent in T for the bindings of pred, bindings;
        Return bindings;
      End-If;
    End-If;
  End;
```

Considering that the recognition of helping needs for second type of helping behaviors (i.e., helping what others are doing) is much more complex than that for first type (i.e., back up what other are doing), the above online algorithm implements role-based proactive helping behaviors for the second type. However, the recognition of helping needs for the first type of helping behaviors is comparatively straightforward.

The part for recognition of helping needs in the above algorithms can easily be replaced with such recognition.

V.7 Summary

In this chapter, we have described an efficient representation of shared mental models, role-based shared mental models, and reasoning mechanisms based on role-based shared mental models.

We mainly focused on the representation team process in role-based shared mental models. By taking the advantages of our mechanisms of task decomposition and task delegation, each agent maintains the team process by an individual process and a team organization. The individual process contains the actions related to the agent. The team organization contains agents' shared goals, the role-based plans to achieve the goals, and team structures to perform the plans. Rather than keep a consistent and detailed team process, the team process in role-based shared mental models is only maintained with a low level overlapping, the team organization. A team process in role-based shared mental models is efficient and leads to effective teamwork because it reduces a great amount of unnecessary communication for maintaining consistent team process and facilitates concurrent execution of team processes.

We have explained how role-based shared mental models enable mutual awareness among team members. Based on mutual awareness, two reasoning mechanisms, including role-based proactive information exchange and role-based proactive helping behavior, have been introduced to ensure effective teamwork. We have formalized the conditions under which proactive information exchange and proactive helping behavior takes place. Algorithms for these two reasoning mechanisms also have been given.

CHAPTER VI
ROLE-BASED TEAMWORK ARCHITECTURE AND ITS EVALUATION

In this chapter, we will explain how our teamwork architecture, RoB-CAST (Role-Based Collaborative Agents for Simulating Teamwork), is organized, interprets knowledge in RoB-CAST, represents the knowledge in agents' mental models, coordinates agents, leads their individual actions to achieve team efforts, and conducts reasoning to improve team performance. We will also design an application domain and develop a set of measures of team performance, both domain dependant and domain independent. Experiments will be run to demonstrate that RoB-CAST is flexible in supporting effective teamwork, that proactive helping behaviors in RoB-CAST improve team performance, and that RoB-CAST, based on role-based shared mental models with a low level of overlapping team process, simulates more effective teamwork than other teamwork architectures (based on shared mental models with a high level of overlapping in the team process, such as CAST 3.0). The measurement data, either domain dependent or domain independent, are collected in the experiments for analysis purposes.

VI.1 Role-Based Teamwork Architecture: RoB-CAST

VI.1.1 RoB-CAST Overview

The knowledge of teamwork is specified in RoB-MALLET. The teamwork architecture, RoB-CAST, parses the knowledge in RoB-MALLET and simulates multiple agents that achieve effective teamwork. RoB-CAST supports this goal from the following aspects:

- RoB-CAST allows users to model teamwork in various dynamic and distributed domains. As a domain-independent teamwork language, MALLET provides a set of constructs to specify a variety of knowledge, including individual and mutual beliefs, shared goals, joint intentions, team structures, and domain knowledge.

RoB-MALLET extends MALLET by accommodating concepts related to roles in the specification of team plans, i.e., role-based plans, to specify the knowledge of team processes. Role-based plans are specified in terms of roles and role variables instead of specific agents and agent variables and thus are more flexible and reusable (as we will illustrate in the experiments).

- RoB-CAST can translate the knowledge specified in RoB-MALLET into mental states in the mental models of agents, dynamically decide their actions and evolve their mental states correspondingly, and interact with specific domains. As the knowledge of team processes is specified by role-based plans, RoB-CAST can dynamically decompose the tasks of team processes into individual tasks and delegate them to agents through the mechanisms of role delegation and team assignment. RoB-SMMs accommodate the mechanisms of task decomposition and delegation. Each agent maintains the team process in its RoB-SMM by a team organization and an individual process, which contains a low level of overlapping with the RoB-SMMs of its teammates. RoB-SMMs improve the performance of teamwork from the performance of time and communication.
- RoB-CAST can support various reasoning mechanisms to improve the performance of teamwork. RoB-SMMs can enable mutual awareness among agents. Based on mutual awareness, agents can reason the information needs of other team members and proactively exchange information with the team members; they also can reason the help needs of other team members and proactively provide helping behaviors to other team members.

While the focus of this dissertation is to build a multi-agent system to simulate effective teamwork, RoB-CAST can be used for team training purpose. A training team could be composed of human trainees and software agents. They could interact with each other through training simulations. By incorporating other mechanisms into RoB-CAST, such as coaching mechanisms [70], software agents could communicate and provide coaching instructions to human trainees.

VI.1.2 RoB-CAST Architecture

The RoB-CAST architecture contains a variety of components, which implement the mechanisms introduced in previous chapters, including interpreting knowledge specified by the RoB-MALLET, generating and partitioning responsibilities, role and team delegation, maintaining RoB-SMMs, and reasoning mechanisms based on RoB-SMMs. The RoB-CAST architecture organizes these components as shown in Figure 37. In Figure 37, rectangles represent modules, including the RoB-CAST module components listed below (and shown inside the biggest box in Figure 37) and a simulator and its Java interface; rounded rectangles represent data, including a library of role-based plans, the knowledge input as RoB-MALLET scripts[22], and internal data dynamically generated by RoB-CAST; arrows from data to module components represent that the data are the input of the module components; arrows from module components to data represent that the data are the output of the module components; arrows from module components to module components represent that the module components from which the arrows stem call the module components to which the arrows point. It is important to recognize that each agent has a copy of everything except the simulator, with which all of the agents can communicate. Thus, the activities of the architecture described below take place concurrently in each agent's instance of the architecture.

- The knowledge about teamwork is represented in RoB-MALLET. RoB-MALLET scripts specify operators and positions in domains, agents and their capabilities, teams, beliefs and inference rules, goals, and joint intentions. Team processes are specified by a library of role-based plans. Joint intentions in RoB-MALLET are scripts that specify which agents invoke which role-based plans, i.e., START statements.

[22] To distinguish the RoB-MALLET code for role-based plans from the code for starting execution to achieve a high level goal, we call the former RoB-MALLET code, and the latter RoB-MALLET scripts. The value of this distinction will become clearer when we illustrate dynamically determining a plan to achieve a goal or provide a helping behavior, as we will have to dynamically generate a script to invoke the plan determined.

- The role-based plan parser compiles role-based plans and extracts their plan components, including virtual team, constraints, preconditions, effects, termination conditions, and plan process.
- The RoB-MALLET script parser interprets the knowledge in MALLET and translates it into corresponding mental states. In particular, beliefs and inference rules are represented in the knowledge base, which is implemented by JARE (Java Automatic Reasoning Engine) [41]. If a team of agents invokes a role-based plan via a script, the RoB-MALLET script parser coordinates all involved agents to call their RoB-CAST-PN composers, and the RoB-CAST-PN composers initialize RoB-CAST-PN components for the plan invocation (described in Sec. V.5.3.1) and dynamically load them into their individual processes (described in Sec. V.5.3.3).
- The responsibility generator translates plan processes into team responsibilities (described in Sec. IV.4.3), and the responsibility partitioner decomposes team responsibilities into individual responsibilities (described in Sec. IV.5).
- The RoB-CAST-PN translator transforms individual responsibilities into the representations of RoB-CAST-PN (described in Sec. V.5.3.2).
- The RoB-CAST-PN composer dynamically composes the RoB-CAST-PN of the individual process (described in Sec. V.5.3). The RoB-CAST-PN composer functions as follows:
    - When an invocation is initialized by a RoB-MALLET script (i.e., a START statement), the composer is called bythe RoB-MALLET script parser to load the RoB-CAST-PN components of the invocation into the RoB-CAST-PN of the individual process. After an invocation is done, the composer is called by the executer (described in next item) to finalize the invocation by removing the RoB-CAST-PN components of the invocation from the RoB-CAST-PN of the individual process. The functionality of the RoB-CAST-PN composer described in this paragraph is for dynamically initializing or finalizing the

RoB-CAST-PN of the individual responsibility for START statements in RoB-MALLET.

- When the executor fires a plan transition, the composer is called by the executor to expand the plan transition with the representation of RoB-CAST-PN corresponding to the responsibilities of the roles delegated to the agent according to the team organization. After the plan is done, it is called to shrink the representation of RoB-CAST-PN back to the plan transition. As explained in Chapter V, the firing of a plan transition only dynamically loads the RoB-CAST-PN components corresponding to the plan; the actual execution of the plan is through executing the RoB-CAST-PN components by the executor. The functionality of the RoB-CAST-PN composer described in this paragraph is exactly for dynamically composing the RoB-CAST-PN of the individual responsibility for firing plan transitions.
- When the executor fires a selection transition and correspondingly a role delegated to the agent is selected to fill a role variable, the composer is called by the executor to load the representation of RoB-CAST-PN corresponding to the responsibilities of the role variable. Afterwards, when the executor fires the same selection transition again and another role delegated to anther agent is selected, the composer is called by the executor to remove the previsouly-loaded representation of RoB-CAST-PN from the individual process.

- The executer keeps executing the RoB-CAST-PN of the individual process (described in Sec. V.5.5). When a token is put in an out-proxy place, it sends the token to its corresponding in-proxy place in the individual process of another agent according to the team organization. If an operator transition fires, it drives the simulator to perform the corresponding operation through the domain interface driver. If a plan transition fires, it calls the team assignment maker to search an admissible team assignment and the RoB-CAST-PN composer to expand the plan transition. After all the transitions corresponding to the plan

invocation have fired, it calls the RoB-CAST-PN composer to shrink all the RoB-CAST-PN components corresponding to the plan invocation back to the plan transition. If a selection transition fires, it calls the team assignment maker to search a role to fill the role variable and calls the RoB-CAST-PN composer to load the representation of RoB-CAST-PN corresponding to the responsibility of the role variable to the agent to which the role is delegated. Afterwards, if the selection transition fires again and another role delegated to another agent is selected, it calls the RoB-CAST-PN composer to remove the representation of RoB-CAST-PN for the role variable from the first role's agent, and then loads the representation of the RoB-CAST-PN corresponding to the role variable responsibility to the agent to which new role is delegated. The executor interacts with the knowledge base: it evaluates conditions, such as preconditions, conditions, and termination conditions, through the knowledge based; and it asserts, retracts and updates belief with the execution.

- The team assignment maker searches admissible team assignments and roles for the role variable selections (described in Sec. IV.6.4). It also coordinates agents to record team assignments and role variable selections in their team organization. The team assignment maker is called by the executor when a plan transition fires or when a selection transition fires. After an admissible team assignments is found or a role is selected for the role variable, the executor calls the RoB-CAST-PN composer to corresponding compose the RoB-CAST-PN of the individual process according to the admissible team assignment or the result of the role selection.
- The information exchanger implements role-based proactive information exchange (described in Sec. V.6.2). It extracts potential information flows from plan processes and dynamically determines the actual information exchange according to the team organization. It then interacts with the knowledge base: when it receives information requests (asks) from other information exchangers, it queries the knowledge base and returns the bindings for the requested

information[23]; when it receives the bindings for information from other information exchangers, it updates the knowledge based correspondingly.

- The helper implements role-based proactive helping behaviors (described in Sec. V.6.3). It extracts potential help needs from plan processes and dynamically determines actual helping behaviors in the form of MALLET scripts according to the team organization. It calls the MALLET script parser to actually trigger the execution of the helping behaviors.
- The domain interface driver drives simulators to performance operations in the actual domains and to obtain data from the domains.

[23] As explained in Chapter V, the requested information is a predicate with variables. The information exchanger queries the knowledge base to find the bindings to the variables, which satisfy the predicate. If such information is not available, the binding is null.

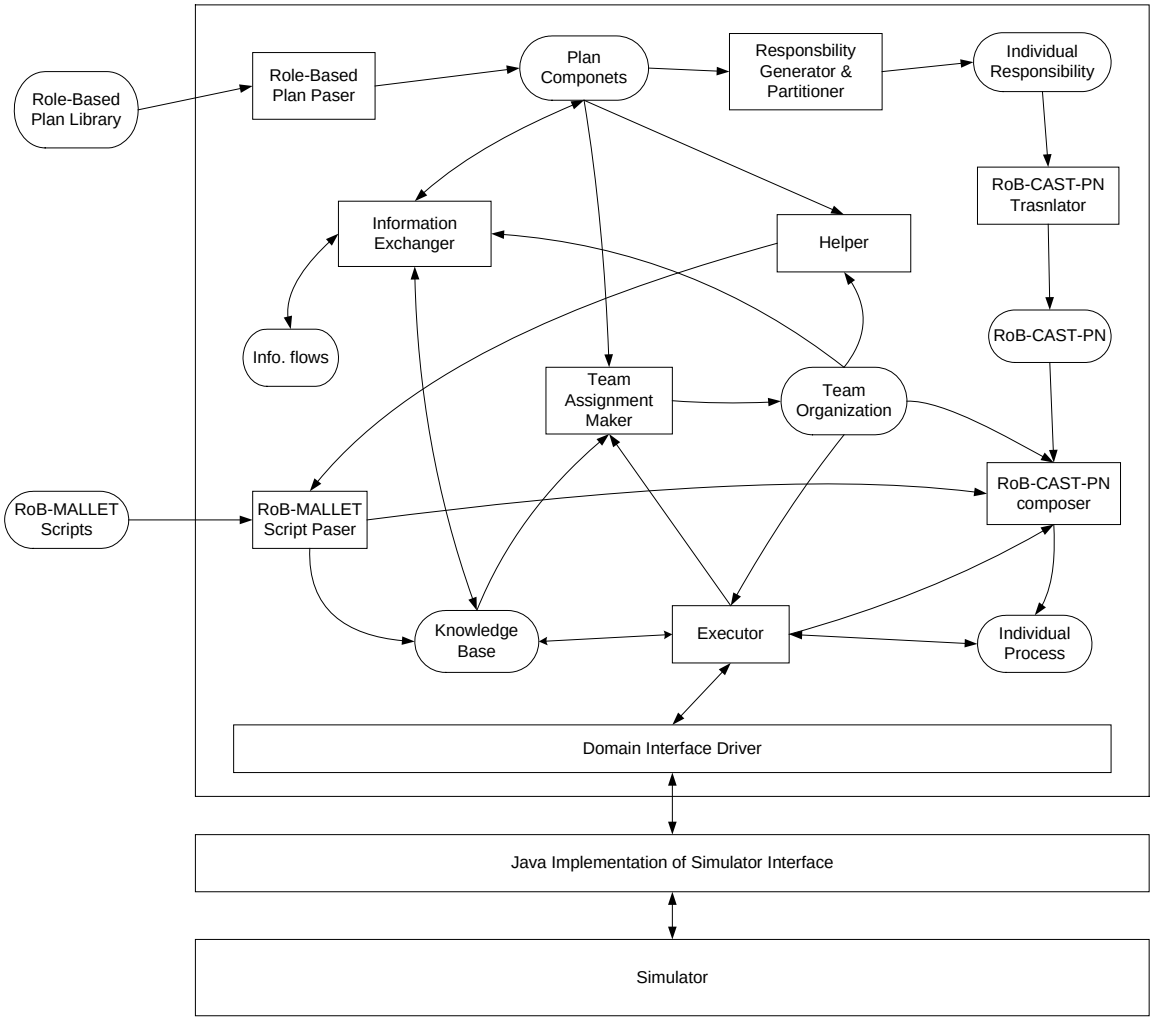

Figure 37. RoB-CAST teamwork architecture.

The RoB-CAST architecture has been implemented in Java with about 270 files and about 20,000 lines of code.

VI.2 Experiment Domain

To evaluate the RoB-CAST architecture, experiments need to run using the agents supported by the RoB-CAST architecture. Many domains, such as Robocup soccer simulator, Space Fortress [14, 89, 116], and AWACS, could be used as candidate

domains. For the purpose of evaluating the RoB-CAST architecture, a proper domain should have the following properties:

- It contains multiple agents;
- Agents may have different capabilities;
- Agents can communicate with each other;
- Its tasks require collaborative work of agents, i.e., teamwork. Each agent only plays some role(s) in the teamwork.
- The space of the states in the domain is finite. For example, it only contains a limited number of operators and positions. However, it allows agents conduct various reasoning; and
- It is also applicable to other teamwork architectures to be compared with the RoB-CAST architecture, particularly CAST 3.0.

For these reasons, we choose an extension of the wumpus world described in [86], a multi-agent wumpus world, as the domain for our experiments. A multi-agent wumpus world is a cave of m × n squares surrounded by walls. A square may contain a wumpus or/and a piece of gold. There is a door between any two adjacent squares so that an agent can move from a square to another. However, the agent will be eaten by the wumpus if it moves in a square with a live wumpus. The multi-agent wumpus world extends the wumpus world by introducing multiple agents with different capabilities, such as sense, kill wumpuses, and collect gold. If an agent is capable of sensing, it can smell the stench of wumpus in adjacent squares and thus know which adjacent square contains wumpus, and perceive the glitter of gold and thus know if there is a piece of gold in its square. If an agent is capable of killing wumpus, it can shoot toward to its adjacent square and kill the wumpus in that square. If an agent is capable of collecting gold, it can collect the gold in the same square.

To specify the knowledge of team processes by role-based plan, a set of positions are defined in the multi-agent wumpus world, including sniffer, fighter and carrier. A sniffer can perform operator sense, a fighter can perform operator kill wumpus, and a carrier can perform operator collect gold.

Experiments are designed to have agents kill wumpuses and collect gold. From the above description, the experiments require teamwork of the agents. Before an agent moves to a square, a sniffer needs to sense that square and to decide if there is a wumpus in the square, and a fighter needs to shoot toward that square if there is. Before an agent collects a piece of gold, a sniffer needs to sense a square and decide if there is a piece of gold in the square. A sniffer may be blocked by squares with wumpuses. A fighter needs to kill the wumpuses in those squares so as to clear a path for the sniffer to exploit squares.

We will use a multi-agent wumpus world (described in Sec. VI.4) in a set of experiments. Some variants (described in Sec. VI.4.1, 0 and VI.4.3) may be made in different experiment for different evaluations.

VI.3 Measures

To evaluate the RoB-CAST architecture, we collect the data in a variety of measures of team performance, either domain dependent or domain independent. To avoid possible confusion on the terms of domain dependent and domain independent measures, we note that the two terms are named from the perspective of collecting data. Domain dependent measures mean that their data only can be collected in specific domains while domain independent measures mean that their data can be collected in any domain.

The domain dependent measures include

- the number of wumpuses killed,
- and the amount of gold collected.

These measures are related to our experiment domain, a multi-agent wumpus world. If other domains are used for experiments, the domain dependent measures vary with the domain. In our experiments, we may focus on different domain dependent measures for different evaluations.

Defining completely domain independent measures that are meaningful across every domain is not as useful as defining domain independent concepts that can be applied to specific domains. For example, one can use the domain independent concept of time

measurement and apply it to the measurement of time for benchmark tasks in specific domains. It is in this domain independent concept sense that we define addition performance measures. The generic categories of measures we define are time performance and communication performance. As noted above, the data for these can be collected in a domain independent manner.

We measure time performance by the absolute execution time of a plan, i.e. the duration from starting to finishing a plan for achieving a goal. There are multiple effects contributing to the time performance, the operation times, the overhead of an architecture, and the concurrency support of the architecture. The former is clearly a function of the domain, but if comparisons are being made among architectures, should impact time measurements in the same way. As shall be shown, by making the operation times significant, differences in handling of concurrency can be shown. By making the operation times negligible, system overhead can be shown. We shall do both. Ideally, a teamwork architecture should have low overhead and good concurrency support. The latter allows agents to finish plans faster and give them more time for reasoning to improve teamwork, such as proactive helping behaviors.

Communication performance is important for a teamwork architecture, too. An efficient teamwork should minimize the amount of communication to achieve teamwork. Moreover, in a mixed team of human subjects and software agents powered by a teamwork architecture, an inefficient teamwork architecture may generate a great amount of communication, which is beyond human capability of message processing.

Communication can be classified into different categories: communication for maintaining team processes, communication for proactive information exchange, and other communication, such as the communication for proactive helping behaviors. In RoB-CAST, the communication for maintaining team processes includes communication for maintaining team organizations, communication for passing tokens between pairs of proxy places and communication for coordinating the firings of special transitions (including plan, selection, and achieve transitions) during the execution of individual

processes. In CAST 3.0, the communication for maintaining team processes is mainly for maintaining consistent copies of CAST-PNs for agents.

These measures are common measures of team performance in different experiment domains. They may be affected by specific domains. For example, operators may have different durations and thus the amount of time taken is different in different domains. However, given a specific plan on a specific domain, the differences on these measures between different teamwork architectures are caused by the mechanisms in the teamwork architectures. So, comparing these measures between different teamwork architectures can expose how efficiently the architectures support teamwork.

RoB-CAST supports flexible teamwork in addition to effective teamwork, including plan reusability and simultaneity of invocation. Plan reusability means that a role-based plan can be used by different teams of agents to achieve their goals. These different teams of agents vary in the number of agents and the capabilities of agents. Simultaneity of invocation means that agents can be involved in multiple independent goals/plans (i.e. some goal/plan is not a subgoal/subplan of any other goal/plan) at a time.

VI.4 Experiments and Analyses

Using Java we have implemented a simulator of the multi-agent wumpus world and its interface. Every agent is empowered by the RoB-CAST architecture or the CAST 3.0 architecture for comparison purposes. All agents interact with the simulator of the multi-agent wumpus world through its interface. We have conducted three experiments on the multi-agent wumpus world for evaluating different aspects of our RoB-CAST architecture. Experiment 1 will illustrate that the RoB-CAST/RoB-MALLET architecture is flexible enough to allow reuse of role-based plans by running a role-based plan with different formation of agents. Experiment 2 will illustrate that RoB-CAST supports proactive helping behaviors and that proactive helping behaviors improve team performance by comparing two the identical teams except that one is empowered with the mechanism of role-based proactive helping behavior while another is not. In addition, experiment 2 will also illustrate that the RoB-CAST is flexible to support simultaneous plan invocations at the top level. Experiment 3 will illustrate how

efficiently our RoB-CAST architecture can support effective teamwork by comparing it with CAST 3.0.

We would like to note that the simulator of the multi-agent wumpus world used in one experiment may be slightly different from that used in another. In particular, the sets of operators may be different and some operators may be implemented differently. We will note those differences and their rationale when we describe the experiments.

VI.4.1 Experiment 1 and Analysis

We want to demonstrate the flexibility of the RoB-CAST architecture in supporting teamwork from two aspects: plan reusability and simultaneity of invocations. Plan reusability means that a role-based can be used by different formations of teams in which there are different number of agents with different capabilities. Simultaneity of invocation means that an agent may be involved in multiple plan invocations (at the top level) dynamically and simultaneously. In experiment 1, we will demonstrate that the RoB-CAST architecture can support plan reusability. Later in experiment 2 described in Sec. 0, we will demonstrate that the RoB-CAST architecture can support simultaneity of invocation as well as that proactive helping behaviors can improve the team performance.

In experiment 1, we have five teams of agents that could invoke a role-based plan *scankillpick* (described in APPENDIX D) in a wumpus world, which is randomly generated. Figure 38 is an example of 100 squares (10 by 10) of a randomly generated wumpus world. The squares contain 20 wumpuses and 20 pieces of gold. The teamwork specified in plan *scankillpick* is to scan wumpuses and gold, to kill found wumpuses and to collect found gold in the wumpus world. Initially, the agents, which invoke plan *scankillpick*, have no knowledge of the squares, i.e., which square contains a wumpus and/or a piece of gold.

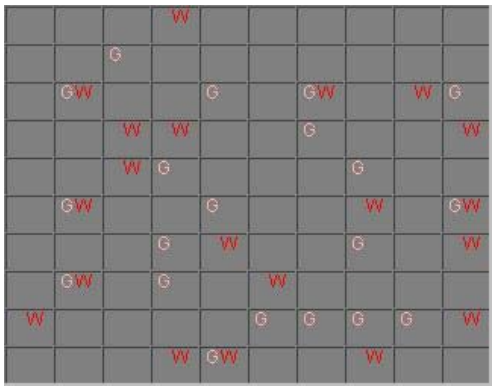

Figure 38. The wumpus world used in experiment 1 and 3.

In APPENDIX C, all operators used in the experiment are specified. The agents take time to accomplish these operators as in our real life. The durations of operators, including move, sense, shoot and collect, are listed in Table 1.

We note that we have added some operators, including `startClock`, `selectWumpusToKill`, `selectGoldToCollect`, `selectSquareToDetect`, `getLock` and `freeLock`, in the operator declarations for collecting data and easier expression. Operator `startClock` is used to have an agent start collecting measurement data. Operators `selectWumpusToKill`, `selectGoldToCollect`, and `selectSquareToDetect` are added to utilize the precondition capabilities of MALLET and RoB-MALLET to block execution until the necessary conditions (e.g. that a wumpus has been found) are satisfied.

Table 1. The durations of operators in experiment 1 and 3

Operator	Duration (ms)
move	200
updateMap	200
sense	100
shoot	100
collect	100

Operators `getLock` and `freeLock` are used to ensure that an agent only detects one target square, kills one wumpus, or collects one piece of gold at a time. If an agent were to do more than one of the tasks at a time, the agent would have to move to more than one target square simultaneously and thus the tasks would interfere with each other. To avoid such interference, we create a lock for each agent and have the agent get the lock before performing any of the operations that might cause interference and release its lock after doing them.

In later experiment 2 and 3, all roles in the top level of plans are delegated to different agents and thus `(targetloc r x y)` is not a critical section. So, we will not use operators `getLock` and `freeLock`. However, we will also use operators, `selectWumpusToKill`, `selectGoldToCollect` and `selectSquareToDetect` for the reason that we just explained. Because `(targetloc r x y)` is no longer a critical section, we let these three operators assert belief `(targetloc r x y)` directly, rather than assert temporary beliefs `(killing r x y)`, `(collecting r x y)` and `(detecting r x y)` and then change them to `(targetloc r x y)` respectively.

All the operators above are only for convenience and not actual operators in the domain. To execute these operators, a teamwork architecture just calls them without actually interacting with the wumpus domain. These operators can be executed instantaneously by the teamwork architecture. In order to avoid any side effect of introducing the operators, we set the duration of these operators to be zero and we do not count these operators in our measurement data.

The knowledge for the five teams is listed in APPENDIX D. We have these teams invoke plan `scankillpick` separately. The structures of the teams, the capabilities of the agents in the teams, and team assignments for the teams invoking plan `scankillpick` are listed in Table 2. We would note that we have run plan `scankillpick` with many formations of teams in addition to those the listed teams. The teams listed in Table 2 include all possible combinations of team assignments with minimum capabilities. If the agents in a team have fewer capabilities, they fail to execute

plan `scankillpick`. If the agents in a team have more capabilities, they execute plan `scankillpick`.

Table 2. Teams with different formations executing plan *scankillpick* and team assignments

Team	Agent	Capability	Role(s) in ***scankillpick***
T1	ag1	`startClock move sense selectSquareToDetect shoot selectWumpusToKill collect selectGoldToCollect getLock freeLock`	r1, r2 and r3
T2	ag2	`startClock move sense selectSquareToDetect getLock freeLock`	r1
	ag3	`startClock move shoot selectWumpusToKill collect selectGoldToCollect getLock freeLock`	r2 and r3
T3	ag4	`startClock move shoot selectWumpusToKill getLock freeLock`	r2
	ag5	`startClock move sense selectSquareToDetect collect selectGoldToCollect getLock freeLock`	r1 and r3
T4	ag6	`startClock move collect selectGoldToCollect getLock freeLock`	r3
	ag7	`startClock move sense selectSquareToDetect shoot selectWumpusToKill getLock freeLock`	r1 and r2
T5	ag8	`startClock move sense selectSquareToDetect getLock freeLock`	r1
	ag9	`startClock move shoot selectWumpusToKill getLock freeLock`	r2
	ag10	`startClock move collect selectGoldToCollect getLock freeLock`	r3

Table 3 shows that the five teams have successfully completed the teamwork specified in plan `scankillpick`. Even though the teams have different time performances of accomplishing the teamwork, the RoB-CAST architecture is flexible for different formations of teams to invoke the role-based plan `scankillpick`. We note

that, each of teams T2, T3, and T3 consists of two agents, however, the two agents in different teams play different roles. The workload of the agent taking r1 (sniffer) directly decides team performance. The faster that agent finds wumpuses and gold, the faster the team finishes the teamwork. In T2, the agent taking r1 did not take another role. In T3 and T4, the agents taking r1 also took another role (r2 in T3 and r3 in T4). Consequently, the team performance of T2 is better than those of T3 and T4. For the similar reasons, T3 killed wumpuses slower than T4 because the agent taking r2 did not take other role in T3 while the agent taking r2 took r1; and T3 collected gold faster than T4 because the agent taking r3 did not take other role in T3 while the agent taking r3 took r1.

Table 3. The duration of different teams executing plan *scankillpick*

Team	Time to kill all wumpuses (s)	Time to collect all gold (s)
T1	387	429
T2	151.5	183
T3	285	241
T4	253	296.5
T5	139.5	163.5

VI.4.2 Experiment 2 and Analysis

In experiment 2, we will demonstrate that RoB-CAST can enable proactive helping behaviors to improve the team performance and that RoB-CAST is flexible in supporting simultaneity of invocation. In experiment 2, we consider two teams, with a total of three agents, pursuing different top level goals. Team T1 consists of two agents, which scan a wumpus world and collect the gold found. Team T2 consists of agent ag2 only and has a separate goal that is irrelevant to the main point of this example. We will represent it simply by random motion through the world. The important feature is that ag2 is able to kill wumpuses. We will show the performance of T1 with and without helping behavior from T2. In this experiment, no value, per se, is attached to the killing of wumpuses. Its

goal is simply to collect as much gold as possible as quickly as possible. The evaluation is based on the comparing the measurements of team performance, collected by the agents with proactive helping behaviors, with those measurements, collected by the agents without proactive helping behaviors.

In this experiment, we use a world (Figure 39) with 400 squares (20 by 20). The squares contain 40 wumpuses and 60 pieces of gold. Rather than randomly generate the wumpus world, however, it has been specifically constructed to contain some interesting scenarios and allow many helping behaviors. A piece of gold may be surrounded by wumpuses. For example, a piece of gold in the corner is surrounded by two wumpuses, a piece of gold at the edge is surrounded by three wumpuses, and a piece of gold in the middle is surrounded by four wumpuses. Such gold is unreachable unless a path is opened by killing one of the surrounding wumpus. Also, a piece of gold may be in a square together with a wumpus. Such gold is also unreachable unless the wumpus is killed. These are 25 pieces of unreachable gold in the wumpus world as shown in Figure 39. Initially, the agents have no knowledge of the squares.

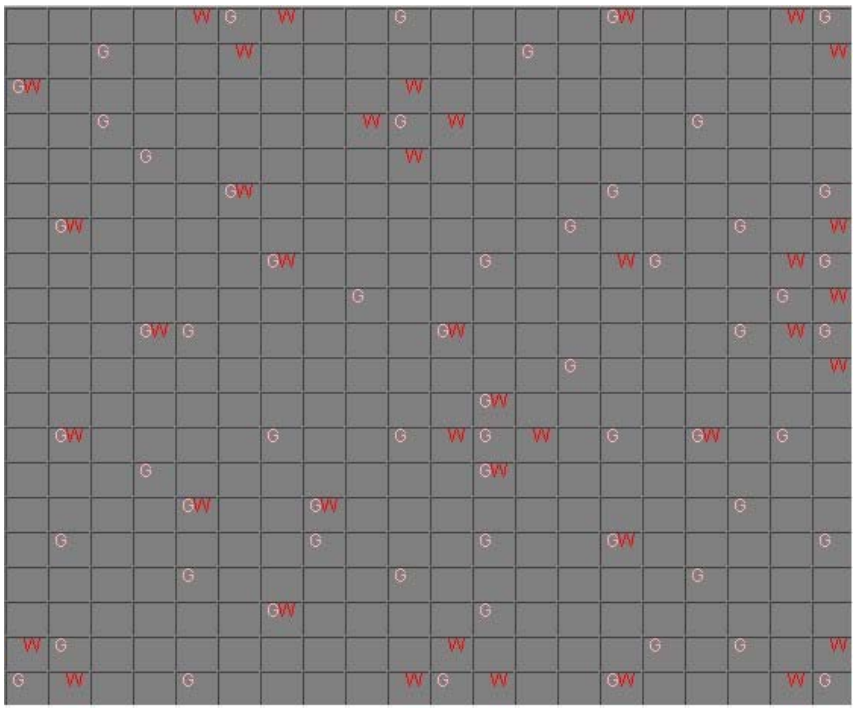

Figure 39. The wumpus world used in experiment 2.

The agents are capable of different sets of operators. The operators are the same as those in experiment 1 unless we note differently. Agent ag1 can perform actions move and sense, i.e., ag1 is a sniffer. The action of sense is different from that in experiment 1. If ag1 performs actions sense, it can detect the gold and the wumpuses in the squares of radius 2. Agent ag2 can perform actions randmove, move and shoot, i.e.. Ag2 is a fighter. The operator move uses a probability distribution weight more heavily in the direction of the target as a basis for randomly selecting the next destination[24]; however, it will not pick a location in which a wumpus is residing. The operator randmove is similar, except that the probability distribution used to select the point is not weighted in any direction. It is intended that this will be executed when an agent is moving, but not toward any particular target. Agent ag3 can perform actions move, movein, and collect. The operator movein is used to move to a specific adjacent square. The agent successfully moves to the square if there is no wumpus at the square; otherwise, it waits until there is no wmpus in the target square and then moves to the square, or it fails after it waits for a certain time. Table 4 shows the durations of the actions.

Table 4. The durations of actions in experiment 2

Operator	Duration (ms)
move	200
movein	200
randmove	200
sense	100
shoot	100
collect	100

[24] We note that operator move essentially has non-deterministic effects. RoB-MALLET and MALLET both use a conjunction of conditions to represent an effect and thus they are not convinent to represent non-deterministic effects. In our experiments, we asserted non-deterministic effects in the domain interfaces, i.e., the interface between RoB-CAST and the Wumpus world. Being more specific, when the executor of RoB-CAST calls the operator through the interface, the interface not only calls the implementation of the operator in the domain, but also asserts the non-deterministic effects.

In APPENDIX E, all operators used in the experiment are specified. Similar to experiment 1, we add operators `startClock`, `selectWumpusToKill`, `selectGoldToCollect`, and `selectSquareToDetect`, in the operator declarations. An additional operator checkSurroundingWumpus is added to check if the performer believes a square is surrounded by wumpuses. It is invoked only by the carrier. This operator is part of a sequence that establishes a need for killing a wumpus that ag2 can detect. The duration of these operators are zero too and we do not count these operators in our measurement data.

A goal of team T1 is to collect as many pieces of gold as possible. The agents start their teamwork from square (1, 1), i.e., the square in the left top corner. Agent ag1 keeps moving and detecting wumpuses and gold; agent ag3 moves to the gold and collects it once it knows a piece of gold.

Agent ag2 is not directly involved in collecting gold. We emulate the unrelated activity in which ag2 is involved just by having ag2 randomly move. The agents are allowed to communicate (through proactive information exchange) with each other to share their knowledge about the squares. We note that ag3 behaves differently in the situations under which the gold to be collected is reachable from its behavior in the situations under which the gold to be collected is unreachable (these situations imply help needs). The decision making for different situations is included in the plan used. However, the plan does not include any statement to let ag2 invoke any facility of decision making to decide whether or not to help ag3. The teams with and without helping behaviors use the same plans. As we have discussed in Chapter V, our goal is to provide the mechanism by which helping behaviors can automatically take place, not the specific decision making tools to decide whether or not to help. Consequently, for purposes of illustrating the helping behaviors, there is no loss of generality in experiment 2.

The agents are empowered by the RoB-CAST architecture both with and without proactive helping behaviors. The knowledge represented in RoB-MALLET, including

the strategy of the teamwork described in the above paragraph, is listed in APPENDIX F.

The agents with and without proactive helping behavior behave differently in these situations:

- once ag3 knows a piece of gold surrounded by wumpuses, ag3 tries to find a path to reach the gold; or
- once ag3 knows a piece of gold with a wumpus, ag3 tries to move to the square to collect the gold.

Without proactive helping behaviors, nobody kills the wumpus. For safety, ag3 gives up collecting the gold. However, with proactive helping behaviors, ag2 can be aware that ag3 needs help to collect the gold. If ag2 knows it can help ag3 by killing the wumpus; then ag2 provide helping behaviors by starting a plan *kill* to kill the wumpus.

One may question whether providing helping behaviors may jeopardize what agents that would help are doing. As explained in Chapter V, after agents recognize a help need, they invoke planning mechanisms to find a proper course of actions to provide helping behaviors. If a course of actions will jeopardize what the agents are doing, the actions should not been chosen by the planning mechanism for the agent to provide helping behaviors; and the plan mechanism needs to continue to examine other course of actions. If no course of actions is found by the planning mechanism, then the agents cannot help. The mechanism of proactive helping behavior focuses on identifying help needs, finding possible helping plans, and forming teams to invoke the helping plans. Therefore, with acknowledging that many aspects may affect the reasoning about whether agents can find a proper course of actions to provide helping behaviors, in this experiment, we simply assume agents provide helping behaviors once the agents knows a plan which effects can cover the help need and there is an admissible team assignment for the agent to invoke the plan.[25][26]

---

[25] We note that we are aware that the issues discussed in this paragraph are related to helping behaviors, such as prioritizing helping behaviors and on-going tasks and planning proper helping behaviors. We will give more discussions about these issues as future work in Chapter VII. Basically, we

Due to the use of the same teamwork architecture (RoB-CAST) and identical knowledge, the difference between the performance of the teams with and without proactive helping behaviors is only caused by the difference in enabling proactive helping behaviors. We measure their performance corresponding to the goal of the team, i.e., collect as much gold as possible.

The result of the experiment shows that the team with proactive helping behaviors collects 54 pieces of gold, on average, while the team without proactive helping behaviors only collects 35 pieces of gold. Figure 40 shows the distribution of the number of pieces of gold collected over the execution time.

Without proactive helping behaviors, the agents only can collect those pieces of reachable gold (i.e., the gold which is neither surrounded by wumpuses nor in the squares with wumpuses). With proactive helping behaviors, the agents can collect more gold and faster. The agents can collect more gold because the mechanism of proactive helping behaviors can enable ag2 to kill the wumpuses which surround gold or are with gold so that ag3 can move into the squares with gold. The agents can collect gold faster for the reasons:

1. ag3 in the team with proactive helping behaviors gives up fewer pieces of unreachable gold than ag3 in the team without proactive helping behaviors In fact, ag3 in the team without proactive helping behaviors gives up all unreachable gold.
2. ag2 with proactive helping behaviors kills wumpuses so that there are less wumpuses in the map. Consequently, ag1 can move close to undetected squares faster and find gold faster; ag3 can move to found gold faster.

---

propose to weight possible plans for helping behaviors, including to their benefits, side effects and costs, and then apply theoretic decision-making to decide the best plans for helping behaviors.

[26] We also note that our mechanism of Role-Based Proactive Helping behaviors is an automatic mechanism for agents. With this mechanism, users do not need to specify knowledge to direct agents to provide helps. Rather, agents can automatically provide helps to others if needed.

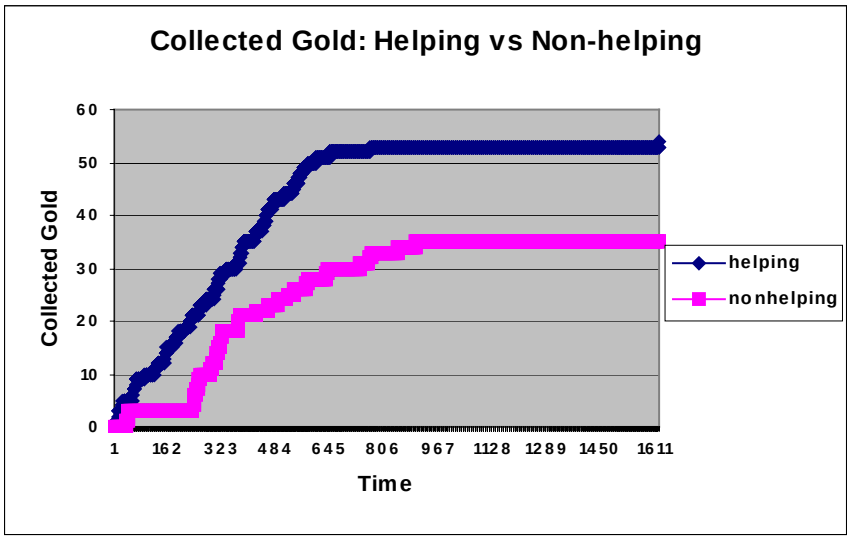

Figure 40. The distribution of the number of pieces of gold collected over the execution time.

As a trade-off, enabling proactive helping behaviors requires extra communication to identify actual help needs as described in Sec. V.6.3. To show the trade-off, we also measure communication performance, including the total number of messages, the number of messages for synchronizing plan execution, the number of messages for information exchange, and the number of messages for helping behaviors. The result of the experiment shows that a great amount of communication is caused by enabling proactive helping behaviors. The reason why the amount of the extra communication is large is, we let the agents check help needs every 30ms so as to responds to help needs promptly. In fact, we can set the frequency of checking help needs much lower and ag2 still can provide helping behaviors promptly, for example, once a second. The extra communication can then drop dramatically. For checking help needs once a second, the amount of extra communication drops to about 1500 messages at end. Most of the extra communication is used to identify actual helping needs. After ag2 identifies help needs of ag3, ag2 starts plan kill to provide helps. Then ag3 can collect gold that originally is unreachable. Consequently team performance is improved. Figure 41 illustrates the distributions of communication cost by agents with and without proactive helping behavior over the execution time.

We also notice that the agents did not collect all gold and some pieces of gold were not collected even though proactive helping behaviors made them reachable. One of the causes of this problem is that, the role-based plan used by the agents specifies that a piece of gold will not be collected once ag3 recognizes the gold is unreachable and later even though the gold becomes reachable later. This can be improved by a better plan, which allows agents to collect the gold, which becomes reachable, but adding this complexity was not necessary to illustrate our point.

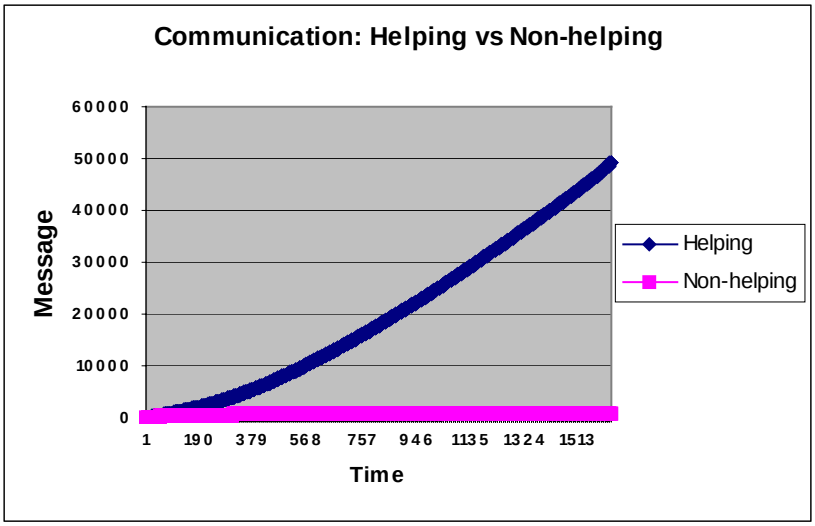

Figure 41. The comparison of communication between with and without proactive helping behaviors.

Another cause is that ag2 did not give up killing the wumpus after ag3 gave up a piece of gold. Fundamentally, ag2 did not terminate its helping behavior after the help need did not exist any more. The reason why our proactive helping behavior does not support this feature is that neither MALLET nor RoB-MALLET[27] can specify the motivation of plan invocation. In other words, START and DO constructs only specify which agents invoke which plans. They do not specify why the agents invoke the plans. Although the termination condition of the plans can stop the plan executions, termination

[27] As explained earlier, RoB-MALLET only extends MALLET by introducing the concepts related to role.

conditions are too general and cannot replace all specific motivations. Some formalism of joint intentions includes motivations, for example the "context" of intent-to and intend-that in Grosz's shared plan theory [36, 37] and the "relative to q" in Cohen's joint intention [22, 59]. MALLET could represent motivations by extending START and DO constructs with a list of conditions as the motivation. For example, START and DO constructs could be extended as:

Start ::= "(" <START> *AgentOrTeamName Invocation* (<FOR> *Cond*+)? ")"

DoMalletProcess ::= "(" <DO> *ByWhomSpec MalletProcess* (<FOR> *Cond*+)? ")"

The semantics of START construct is that, *AgentOrTeamName* starts *Invocation* just because of *Cond*+. If *Cond*+ becomes false during the execution of *Invocation*, the execution is terminated. The semantics of *DoMalletProcess* is that, the performers in *ByWhomSpec* perform *MalletProcess* because of *Cond*+. +. If *Cond*+ becomes false during the execution of *MalletProcess*, the execution is terminated. The only difference between DO and START constructs is that DO statements are used inside plans while START statements are not. Extending DO construct allows us to define more flexible plans, i.e., allowing us to specify the motivations of invoking operators and sub-plans in plans.

We can apply this extension to express the motivation of helping behaviors in this experiment. Agent ag2 with proactive helping behaviors in this experiment starts plan *kill(x, y)* because ag3 wants to perform operator *movein(x, y)* to move to square (x, y) and there is a wumpus in the square. Therefore, we can express that ag2 starts plan *kill(x, y)* for helping ag3 as *(START ag2 (kill x y) FOR (performing ag3 movein x y))*.

We note that T2 was considered a separate team from T1, rather than as a second sub-team of a larger team, to show the more general situation with separate teams. The entire helping behavior scheme works within sub-teams of a team as well.

It is worth making another observation in regard to experiment 2. In experiment 2, two teams start two plans, `randommove` and `scancollect`, simultaneously (at the top level). With proactive helping behaviors, ag2 dynamically finds a plan for a goal (in this case to kill a wumpus by plan `kill`) and builds and executes a script to start it

many times to help ag3 while ag2 is still involved in plan `randommove`. The plans, `randommove, scancollect` and `kill`, are separate team processes for T1 and T2. Agents can proactively exchange information across the boundaries of these three plans. Agents ag1 and ag3 in plan `scancollect` and agent ag2 in plan `randommove` and `kill` can proactively exchange information about whether there are wumpuses in the squares. After ag1 senses a square, ag1 proactively tells ag2 the information about whether there is a wumpus in the square so that ag2 can perform operator randmove in plan `randommove` and move in plan `kill`. After ag2 kills a wumpus in a square by plan `kill`, ag2 proactively tells ag1 and ag3 that there is no wumpus in the square any more. In summary, this illustrates two other features of RoB-CAST:

1) it is flexible in supporting simultaneous plan execution, and
2) proactive information exchange can cross top level plan boundaries so that independent teams doing different things can help each other.

VI.4.3 Experiment 3 and Analysis

Experiment 3 is intended to demonstrate that the RoB-CAST architecture is efficient in supporting teamwork. As explained in Chapter V, RoB-CAST is efficient in supporting teamwork because it uses an efficient representation of shared mental models (i.e., RoB-SMMs). For this purpose, we compare the performance of a team supported by RoB-CAST with that of a logically equivalent team supported by other teamwork architecture (we pick CAST 3.0 because it is the recent teamwork architecture, which is based on shared mental models and supports MALLET). In experiment 3, a team consists of three agents which scan a wumpus world, kill wumpuses found, and collect gold found; the agents' teamwork is tested separately with RoB-CAST and CAST 3.0. The evaluation is based on comparing the measurements of team performance, collected by the agents with RoB-CAST, with measurements collected by the agents with CAST 3.0.

The team, T, consists of ag1, ag2 and ag3. The agents scan wumpuses and gold, kill wumpuses found, and collect gold found in a wumpus world as shown in Figure 38.

Initially, the agents have no knowledge of the squares, i.e., which square contains a wumpus and/or a piece of gold.

The agents are capable of different sets of actions. Agent ag1 can perform actions move, update map, and sense, i.e., ag1 is a sniffer. If ag1 performs action sense, it can detect the gold in its current square and wumpus(es) in adjacent squares. Agent ag2 can performs actions move, update map, and shoot, i.e., ag2 is a fighter. If ag2 shoots, a wumpus in the adjacent square of the shooting direction is killed. Agent ag3 can perform actions move, update map, and collect, i.e., ag3 is a carrier. If an agent moves in to a square with a live wumpus, the wumpus will eat the agent.

The operators, move, sense, shoot and collect, are similar to those used in experiment 1. For the reason explained in experiment 1, we add operators `startClock`, `selectWumpusToKill`, `selectGoldToCollect`, and `selectSquareToDetect`. In APPENDIX G, all operators used in the experiment are specified. Because the syntax of operators in RoB-MALLET is the same as that in MALLET, the same code is used by RoB-CAST and CAST 3.0 for the knowledge of operators.

The goals of the teamwork are to kill all wumpuses and collect all gold. To achieve the goals, the agents start their teamwork from square (1, 1), i.e., the square in the left top corner; agent ag1 keeps moving and detecting wumpuses and gold; agent ag2, once it knows a wumpus location, moves close to the wumpus and shoots it; agent ag3 moves to the goal and collects once it knows a goal. The agents are allowed to communicate with each other through proactive information exchange to share their knowledge about the squares.

The agents are empowered by the RoB-CAST architecture and the CAST 3.0 architecture to perform the teamwork. Although the knowledge about the agents in RoB-CAST is represented in RoB-MALLET and that in CAST 3.0 is in MALLET respectively, they are logically equivalent, including same initial beliefs, the same capabilities, and logically same team plans. The knowledge represented in RoB-MALLET, including the strategy of the teamwork described in the above paragraph, is

listed in APPENDIX H. The knowledge represented in MALLET, including the strategy of the teamwork described in the above paragraph, is listed in APPENDIX I.

Due to the use of logically equivalent knowledge, the team in RoB-CAST and the team in CAST 3.0 should achieve comparable performance. The performance differences between them are only caused by using RoB-CAST and CAST. The performance of the team in RoB-CAST and CAST 3.0 is measure by

- the number of wumpuses killed,
- the amount of gold collected,
- the amount of execution time,
- the total number of messages,
- the number of messages for synchronizing plan execution, and
- the number of messages for proactive information exchange.

To evaluate the efficiency of RoB-CAST and CAST 3.0 in supporting teamwork, we compare the relationships between these measures, particularly between domain independent and domain dependent measures. These relationships include:

- the relationship between the amount of execution time and the number of wumpuses killed ,
- the relationship between the amount of execution time and the amount of gold collected,
- the relationship between the total number of messages and the number of wumpuses killed,
- the relationship between the total number of messages and the amount of gold collected,
- the relationship between the number of messages for synchronizing plan execution and the number of wumpuses killed ,
- the relationship of the number of messages for synchronizing plan execution to the amount of gold collected,
- the relationship between the number of messages for information exchange and the number of wumpuses killed, and

- the relationship between the number of messages for information exchange and the amount of gold collected.

Agents in the experiment can be configured in two modes, distributed and centralized. In the distributed mode, the three agents are run in three platforms and the domain application of wumpus world is run in another platform. In the centralized mode, the three agents are run in a single platform and the domain application of wumpus world is run in another platform. Also, the durations of operators in the wumpus world also can be configured. We run the experiments with two sets of durations, the operators with durations listed in Table 1 and all operators with a duration of 0 *ms*. We have run the experiments in four configurations: 1) distributed mode and operators with durations, 2) centralized mode and operators with durations, 3) distributed mode and operators without durations, and 4) centralized mode and operators without durations.

The comparison of the above relationships in RoB-CAST and in CAST-3.0 with the four configurations are show in Figure 42 - Figure 65. The comparisons of the above relationships in RoB-CAST and in CAST 3.0 with a distributed configuration are shown in Figure 42-Figure 47. The comparisons of the above relationships in RoB-CAST and in CAST 3.0 with a centralized configuration are shown in Figure 48-Figure 53. While we show the relationships in Figure 44, Figure 45, Figure 50 and Figure 51, including the relationship between the number of messages for information exchange and the number of wumpuses killed and the relationship between the number of messages for information exchange and the amount of gold collected, these relationships are shown separately in Figure 46, Figure 47, Figure 52 and Figure 53 for better readability. We can draw the following conclusions based on the comparisons illustrated in these figures:

- According to Figure 42, Figure 48, Figure 54, and Figure 60, we found that, to kill various numbers of wumpuses, the time cost in RoB-CAST is consistently much less time than that in CAST 3.0 for all four configurations. The time cost is somewhat proportional to the number of wumpuses killed in both RoB-CAST and CAST 3.0, however, the slopes are dramatically different and the slope for RoB-CAST is much smaller than that for CAST 3.0.

- According to Figure 43 Figure 49, Figure 49, Figure 55 and Figure 61, we found that in collecting various amounts of gold, the time cost in RoB-CAST is consistently much less time than that in CAST 3.0 for all four configurations. The time cost is somewhat proportional to the amount of gold collected in both RoB-CAST and CAST 3.0, however, the slopes are dramatically different and the slope for RoB-CAST is much smaller than that for CAST 3.0.
- According to Figure 44, Figure 46, Figure 50, Figure 52, Figure 56, Figure 58, Figure 62, and Figure 64, we found that, to kill various numbers of wumpuses, the communication cost in RoB-CAST is consistently extremely less than the communication cost in CAST 3.0 for all four configurations, including the messages for information exchange and synchronizing plan execution. The total number of messages, the number of messages for synchronizing plan execution and the number of messages for information exchange are somewhat proportional to the number of wumpuses killed. However, the slopes of total messages and messages for synchronizing plan execution are dramatically different and the slopes of total messages and messages for synchronizing plan execution for RoB-CAST are dramatically smaller than that for CAST 3.0; and the slope of messages for information exchange in RoB-CAST is about half of that in CAST 3.0. One might wonder why the numbers of message for information exchange in RoB-CAST and CAST3.0 are close for a while and then diverge. The reason is, the information exchange mainly occurred after the sniffer found wumpuses or gold; the sniffer continued to sense undetected squares for a while and thus fewer duplicated information exchanges took place; later, with more revisiting detected squares, duplicated information exchanges took place more often in CAST 3.0.
- According to Figure 45, Figure 47, Figure 51, Figure 53, Figure 57, Figure 59, Figure 63, and Figure 65, we found that, to collect a certain amount of gold, the communication cost in RoB-CAST is consistently extremely less communication than that cost in CAST 3.0 for both configurations, including the messages for

information exchange and synchronizing plan execution. The total number of messages, the number of messages for synchronizing plan execution and the number of messages for information exchange are somewhat proportional to the amount of gold collected. However, the slopes of total messages and messages for synchronizing plan execution are dramatically different. The slopes of total messages and messages for synchronizing plan execution for RoB-CAST are dramatically smaller than that for CAST 3.0; the slope of messages for information exchange in RoB-CAST is about half of that in CAST 3.0. For the same reason as the above, the numbers of messages for information exchange in RoB-CAST and CAST3.0 are close for a while and then diverge.

According to the figures in the first two items of the above list, we found that the ratio of the slope of RoB-CAST to that of CAST 3.0 in configurations 3 and 4 (in which operators have durations) is smaller than the ratio in configurations 1 and 2 (in which operators have durations). In RoB-CAST, the overhead is very small and the execution of operators with durations is about half of the total execution time. However, in CAST 3.0, the overhead is very big. The execution of operators with duration is negligible comparing with the overhead. Therefore, in the configurations 3 and 4, the execution in RoB-CAST becomes faster while execution in CAST 3.0 almost have the same speed as that in configurations 1 and 2. As we can see above, a large amount communication took places in CAST 3.0; consequently, CAST 3.0 took a great amount time to handle the communication. This is the major source of the overhead in CAST 3.0. The essential reason which caused remarkable performance differences in time and communication is that RoB-SMMs in RoB-CAST have a low-level overlapping and thus require less communication to maintain RoB-SMMs, but shared mental models in CAST 3.0 have a high-level overlapping (i.e., every agent maintains a copy of CAST-PN for a team process) and thus requires a great amount of communications to maintain shared mental models. We also found that the total execution time of RoB-CAST in configuration 1 and 2 is even smaller than the total durations of all operator executions. For example, on average, the total execution time for killing 20 wumpusses is about 110 seconds and

there were about 1600 executions of operators with non-zero duration. If the operators are executed sequentially, the total duration should be much more than 110 seconds even though we apply the smallest duration, i.e., 100ms, to the operators. Some operators must been executed concurrently. This shows that RoB-CAST can support concurrent execution.

In summary, RoB-CAST utilizes an efficient representation of shared mental models (RoB-SMMs) and thus it is efficient in supporting effective teamwork. To achieve a certain domain dependent team performance, RoB-CAST is significantly efficient in time and communication. The source of this significant improvement is rooted to the use of role-based shared mental models. We have explained why the role-based shared mental models can improve team performance in Chapter V. The individual processes in our RoB-SMMs maximize the concurrent execution of team process. Also, the representation of team process in our RoB-SMMs has very small overlapping and thus requires much less communication to maintain the consistency of the overlapping. An inefficient representation can cause a great amount of unnecessary communication. For example, CAST 3.0 took a great amount of time to handle the communication for (partial) consistent CAST-PN. This contributed most of the performance degradation of absolute execution time because a great amount of extra time is needed to handle unnecessary communication. This experiment validates our representation of role-based shared mental models can improve team performance.

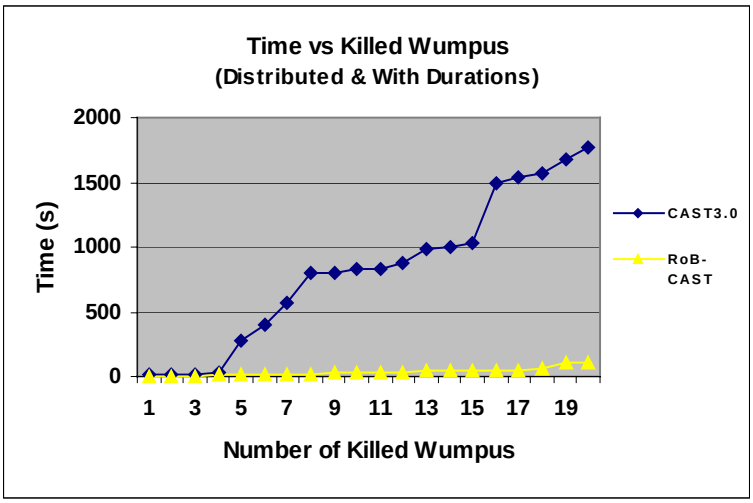

Figure 42. The relationships between the amount of execution time and the number of wumpuses killed in RoB-CAST and CAST 3.0 with configuration 1.

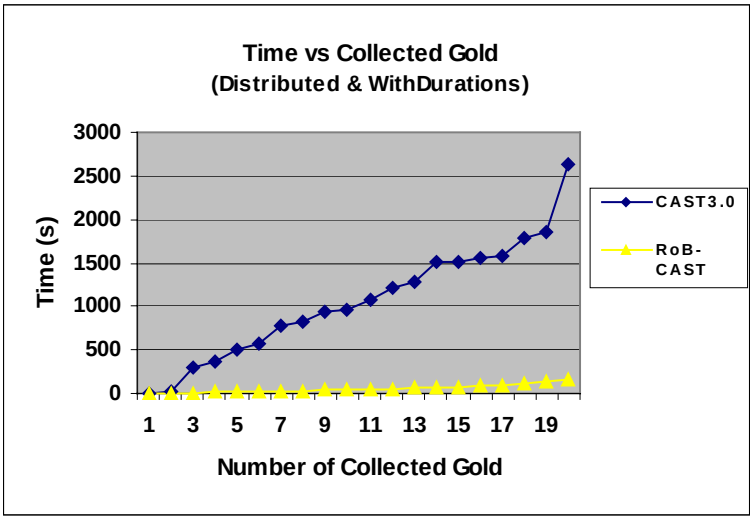

Figure 43. The relationships between the amount of execution time and the amount of gold collected in RoB-CAST and CAST 3.0 with configuration 1.

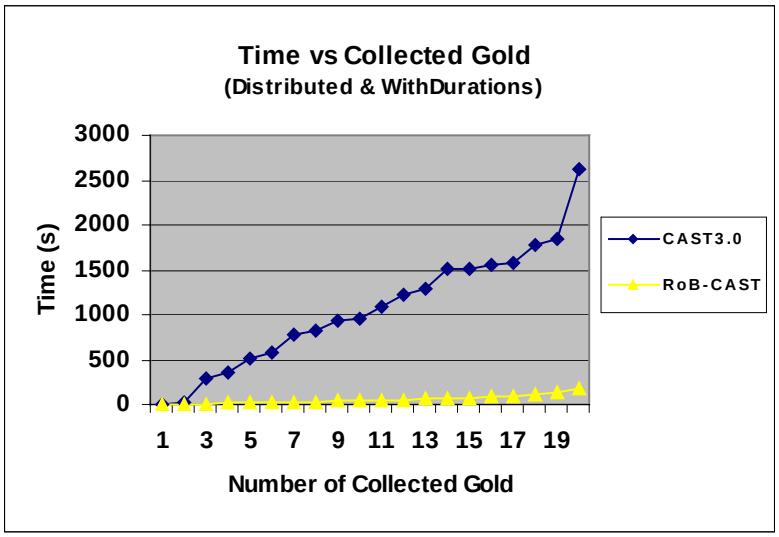

Figure 44. The relationship between the numbers of different types of messages and the number of wumpuses killed in RoB-CAST and CAST 3.0 with configuration 1.

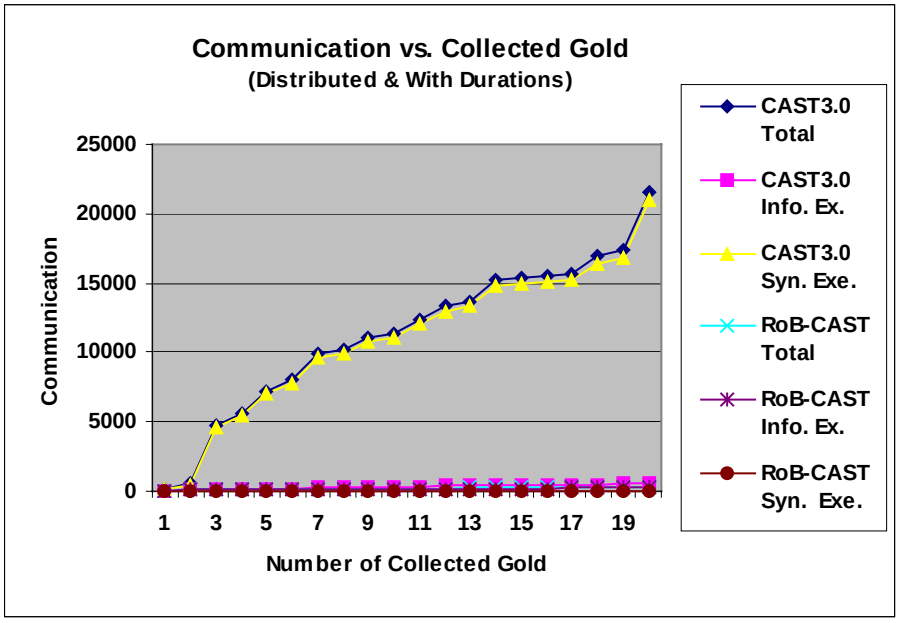

Figure 45. The relationship between the numbers of different types of messages and the amount of gold collected in RoB-CAST and CAST 3.0 with configuration 1.

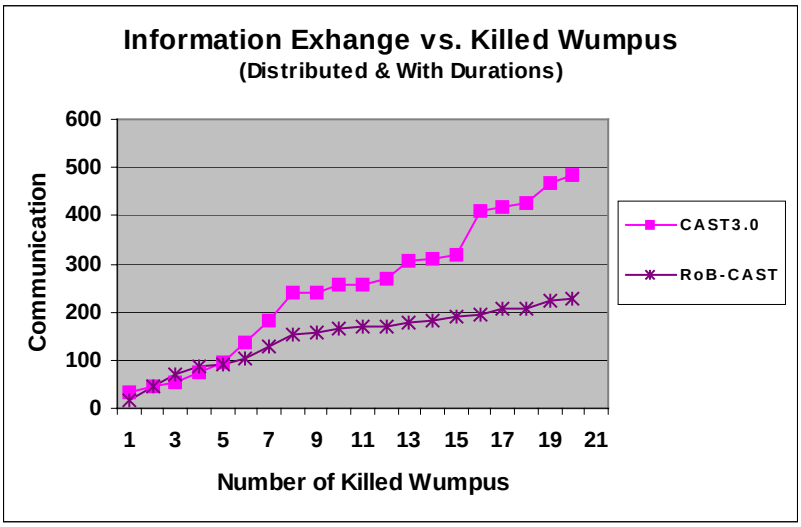

Figure 46. The relationship between the numbers of messages for information exchange and the number of wumpuses killed in RoB-CAST and CAST 3.0 with configuration 1.

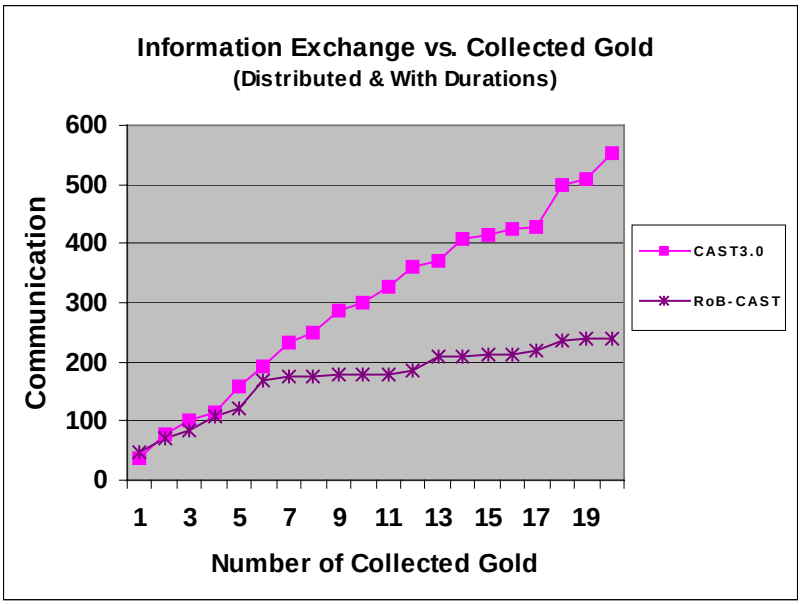

Figure 47. The relationship between the numbers of messages for information exchange and the amount of gold collected in RoB-CAST and CAST 3.0 with a configuration 1.

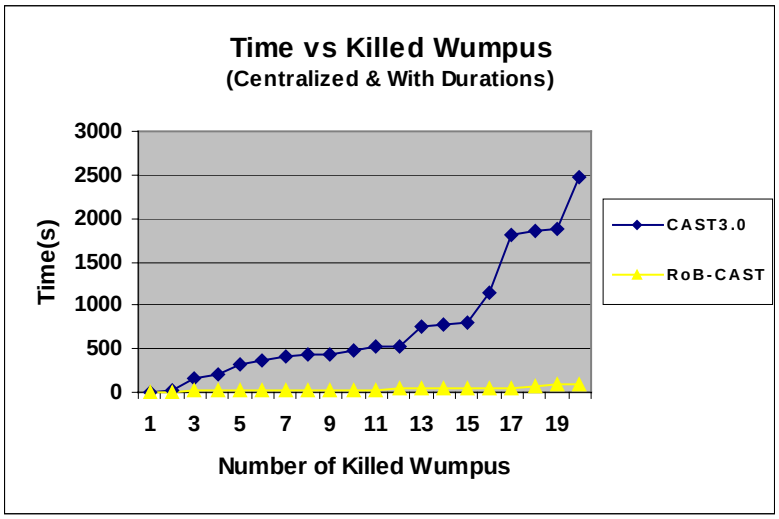

Figure 48. The relationships between the amount of execution time and the number of wumpuses killed in RoB-CAST and CAST 3.0 with configuration 2.

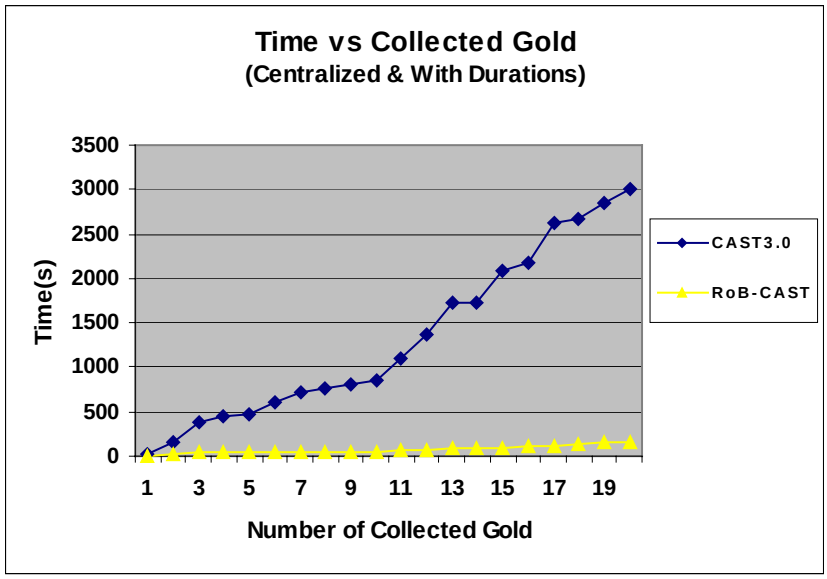

Figure 49. The relationships between the amount of execution time and the amount of gold collected in RoB-CAST and CAST 3.0 with configuration 2.

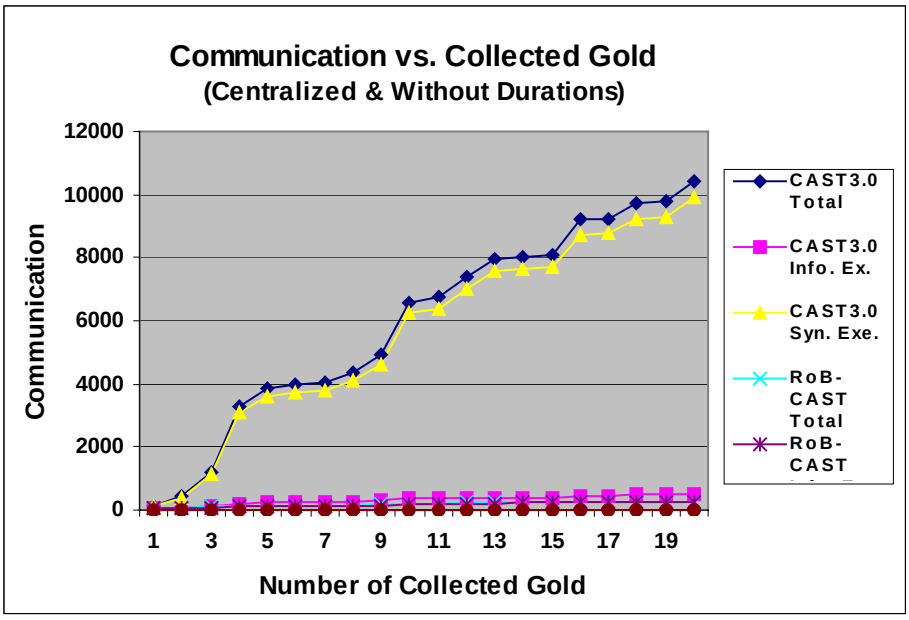

Figure 50. The relationship between the numbers of different types of messages and the number of wumpuses killed in RoB-CAST and CAST 3.0 with configuration 2.

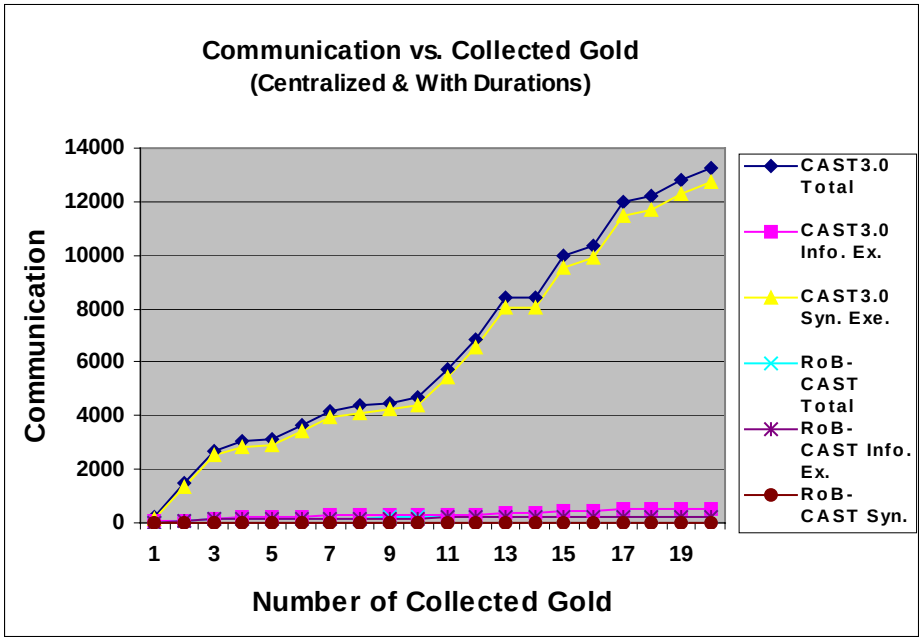

Figure 51. The relationship between the numbers of different types of messages and the amount of gold collected in RoB-CAST and CAST 3.0 with configuration 2.

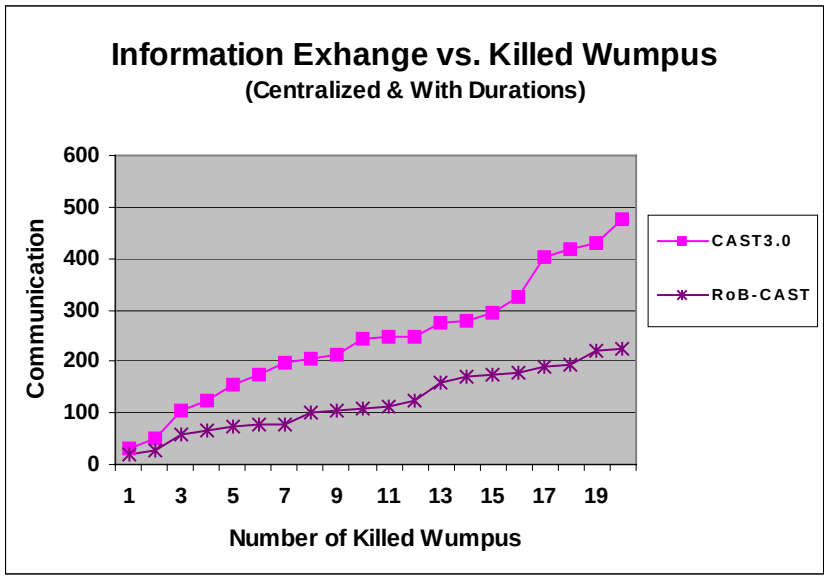

Figure 52. The relationship between the numbers of messages for information exchange and the number of wumpuses killed in RoB-CAST and CAST 3.0 with configuration 2.

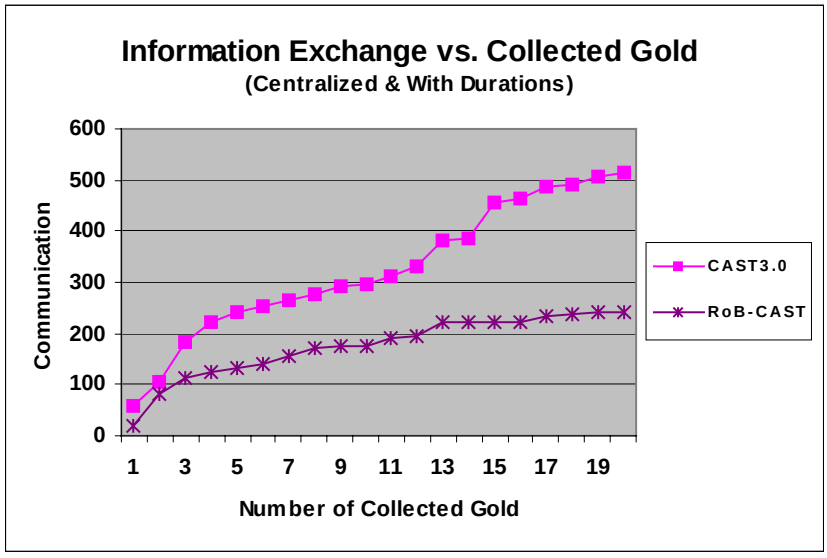

Figure 53. The relationship between the numbers of messages for information exchange and the amount of gold collected in RoB-CAST and CAST 3.0 with configuration 2.

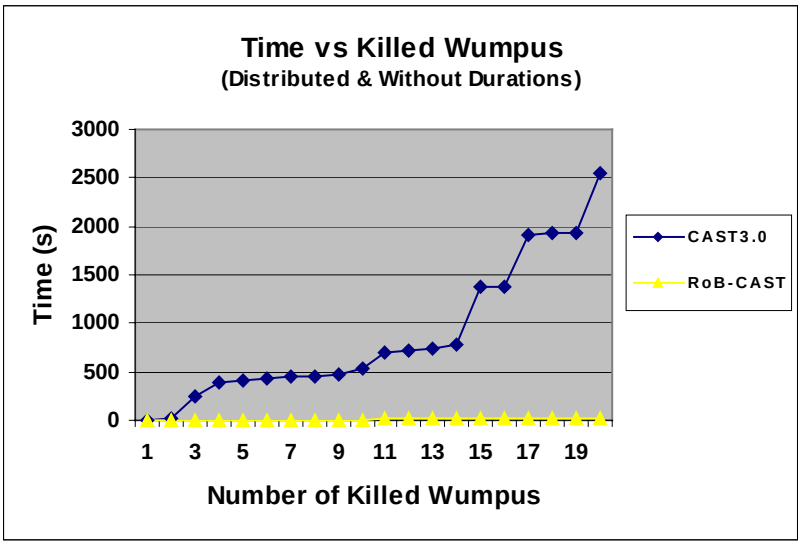

Figure 54. The relationships between the amount of execution time and the number of wumpuses killed in RoB-CAST and CAST 3.0 with configuration 3.

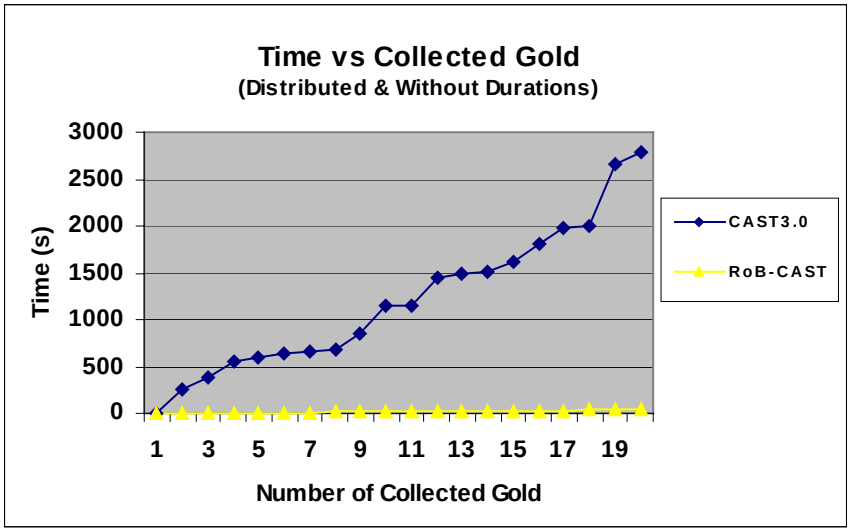

Figure 55. The relationships between the amount of execution time and the amount of gold collected in RoB-CAST and CAST 3.0 with configuration 3.

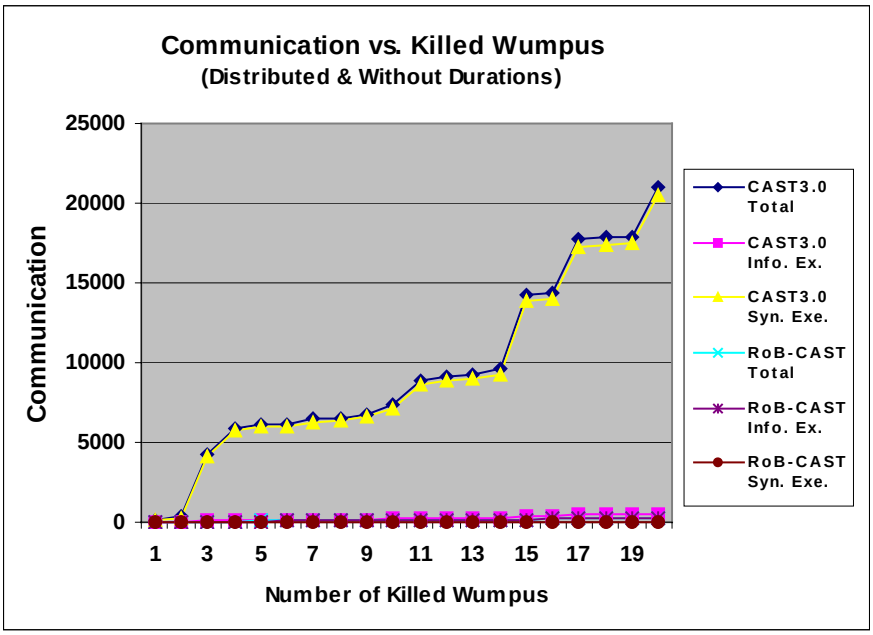

Figure 56. The relationship between the numbers of different types of messages and the number of wumpuses killed in RoB-CAST and CAST 3.0 with configuration 3.

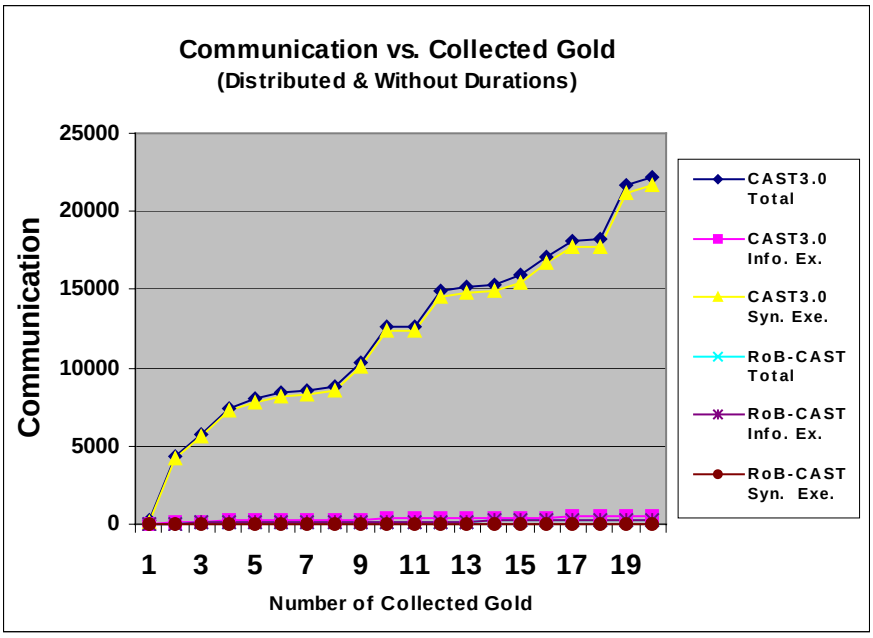

Figure 57. The relationship between the numbers of different types of messages and the amount of gold collected in RoB-CAST and CAST 3.0 with configuration 3.

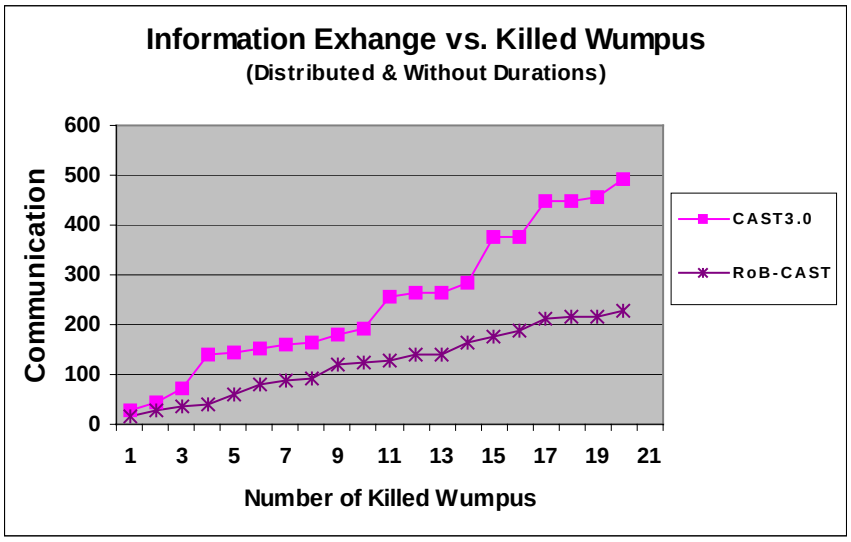

Figure 58. The relationship between the numbers of messages for information exchange and the number of wumpuses killed in RoB-CAST and CAST 3.0 with configuration 3.

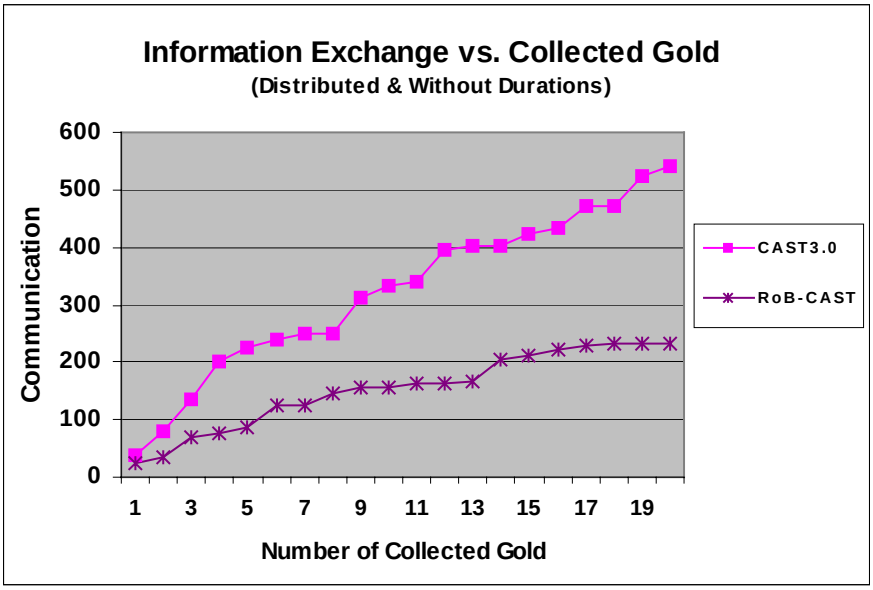

Figure 59. The relationship between the numbers of messages for information exchange and the amount of gold collected in RoB-CAST and CAST 3.0 with a configuration 3.

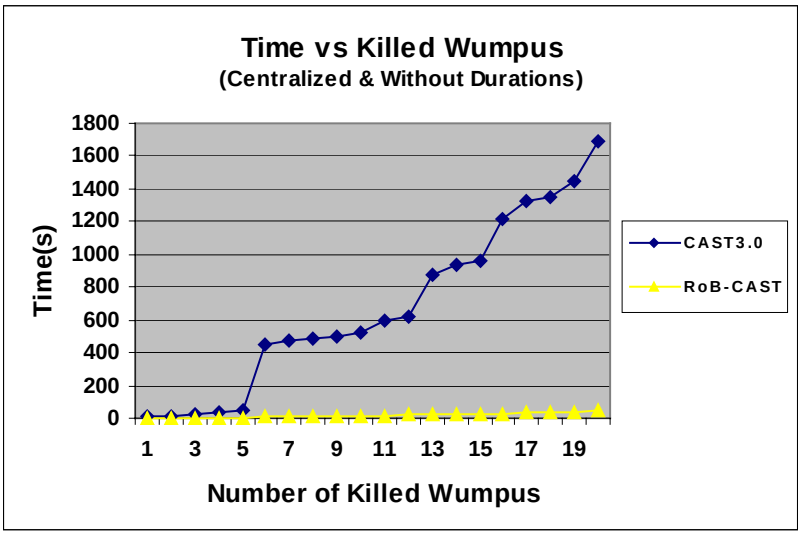

Figure 60. The relationships between the amount of execution time and the number of wumpuses killed in RoB-CAST and CAST 3.0 with configuration 4.

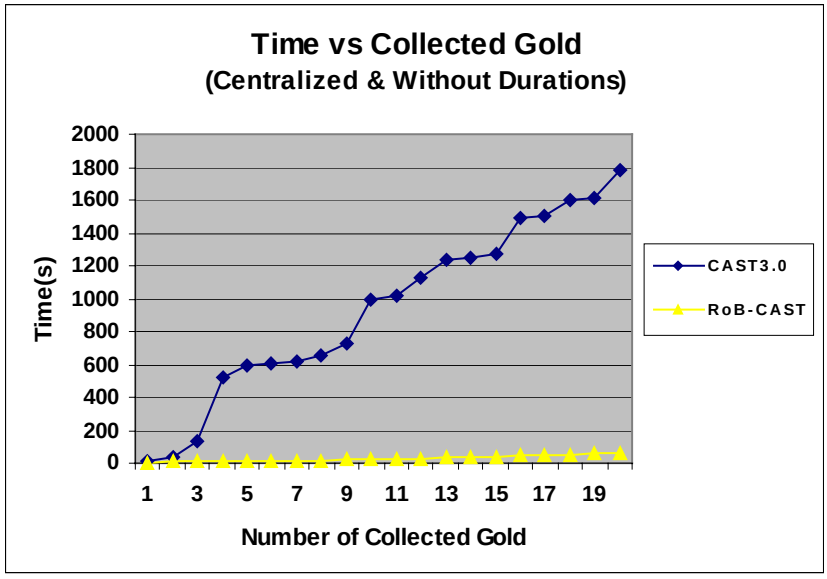

Figure 61. The relationships between the amount of execution time and the amount of gold collected in RoB-CAST and CAST 3.0 with a centralized configuration 4.

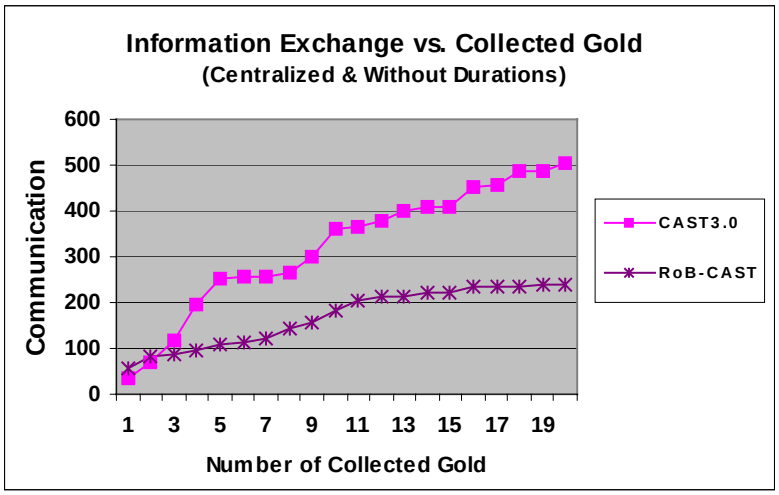

Figure 62. The relationship between the numbers of different types of messages and the number of wumpuses killed in RoB-CAST and CAST 3.0 with configuration 4.

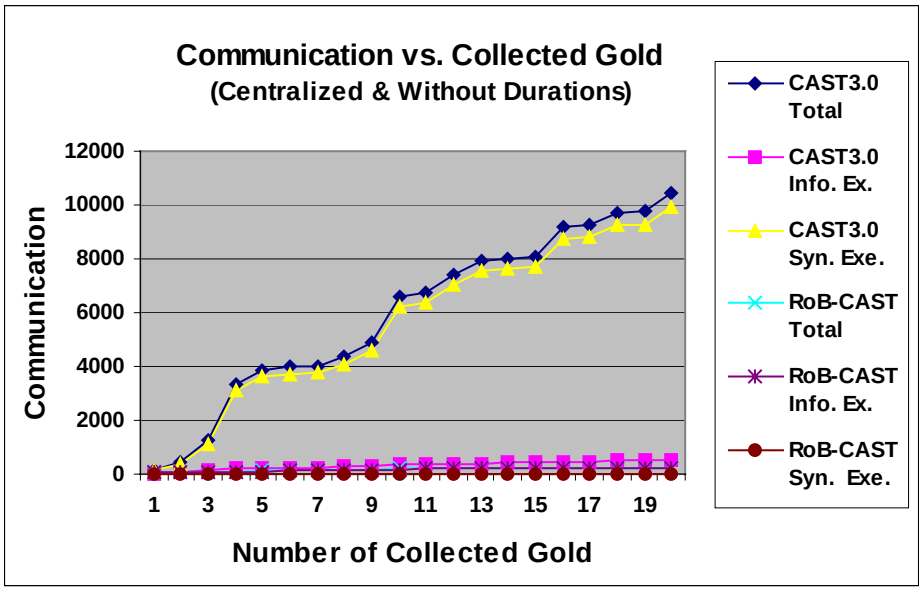

Figure 63. The relationship between the numbers of different types of messages and the amount of gold collected in RoB-CAST and CAST 3.0 with configuration 4.

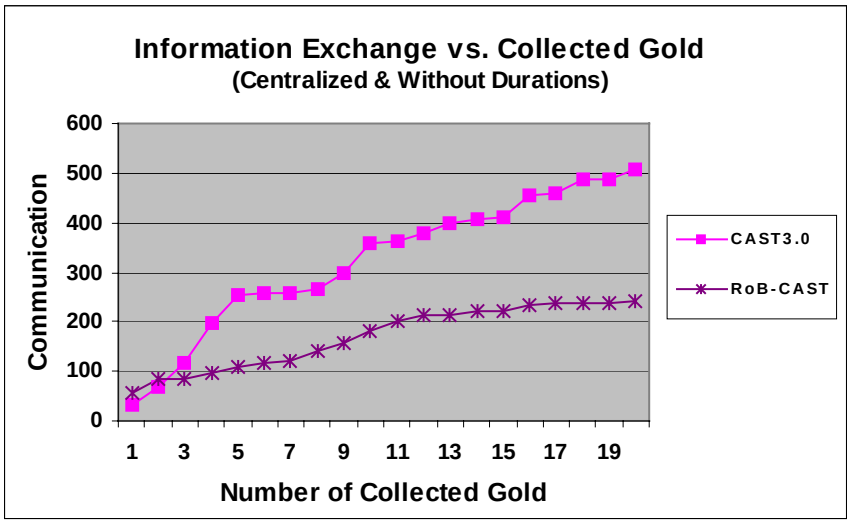

Figure 64. The relationship between the numbers of messages for information exchange and the number of wumpuses killed in RoB-CAST and CAST 3.0 with configuration 4.

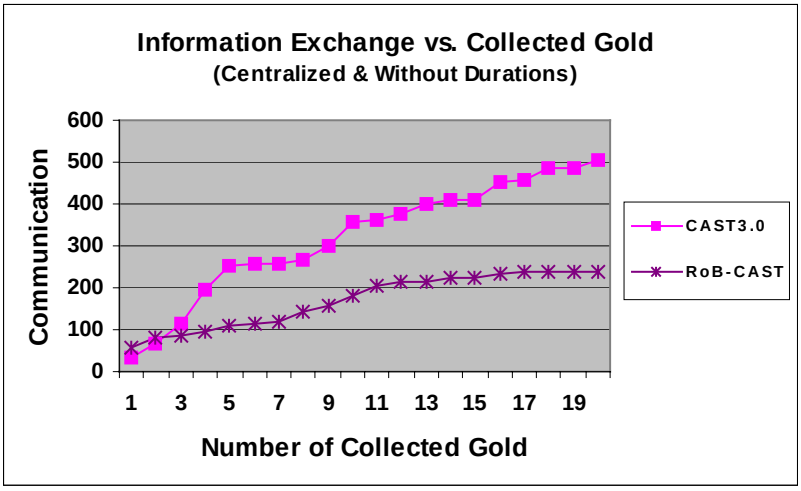

Figure 65. The relationship between the numbers of messages for information exchange and the amount of gold collected in RoB-CAST and CAST 3.0 with configuration 4.

VI.5 Summary

In this chapter, we have described the RoB-CAST architecture and its components. To evaluate the efficiency of the RoB-CAST architecture in supporting effective teamwork, we have chosen a wumpus world as the test bed and designed a set of measures, either dependent or independent of the wumpus world. We have also described three sets of experiments in the wumpus world. Based on the measurement data collected from the experiments and the observation of the experiments, we can conclude that:

- The RoB-CAST architecture is flexible in supporting effective teamwork, including plan reusability and simultaneity of invocation.
- Proactive helping behaviors can improve team performance while extra communication is used.
- The RoB-CAST architecture is efficient in supporting effective teamwork.

CHAPTER VII
CONCLUSIONS AND FUTURE WORK

VII.1 Conclusions

The long-term goal of this research is to build a flexible and effective teamwork architecture, which allows explicitly specifying mental states underlying teamwork, simulates effective teamwork (perhaps a mixed team of software agents and human subjects), allows incorporating various reasoning mechanisms to improve team performance, has planning abilities so as to automatically solves cooperative problems, for general purposes, such as team training, business management, defense simulation, and computer games. In this dissertation, we have made several contributions towards this goal.

First, we have introduced a teamwork programming language, RoB-MALLET, which allows explicit specification of mental states underlying teamwork and conceptual specification of teamwork knowledge for being reused. RoB-MALLET has rich expressivity to explicitly specify the mental states underlying teamwork, such as mutual beliefs, shared goals, joint intentions, operators and team plans. Based on Biddle and Thomas' classificatory concepts of role, the concepts of position, role and role variable has been defined to capture various behavioral aspects of role. Moreover, team plans (i.e., role-based plans in RoB-MALLET) are specified in terms of positions, roles and role variables, instead of specific agent, so as to be reused by different teams of agents. Our experimental results demonstrate that a role-based plan can be used by different teams of agents.

Second, we have developed mechanisms of task decomposition and task delegation in terms of roles and role variables. A responsibility captures the actions to be performed and the temporal order constraints on the actions and characterizes the impact on the mental states of the agent(s) taking the responsibility. A graphical language is also developed to represent responsibilities. A role-based plan is translated into a team responsibility graph and further the team responsibility graph is decomposed into

individual responsibility graphs. By a CSP algorithm, agents in RoB-CAST can dynamically search an admissible team assignment to execute a role-based plan and dynamically select a role (eventually an agent) to fill a role variable. If a role is delegated to an agent, the agent takes the responsibility of the role and thus intends to perform the actions in it. If a role is selected to fill a role variable, the agent to which the role is delegated takes the responsibility and thus intends to perform the actions in it.

Third, we have developed an efficient representation of shared mental models, RoB-SMM, to simulate effective teamwork. In RoB-CAST, each agent maintains a team process by a team organization and an individual process in its RoB-SMM. A team organization records goals, the role-based plans to achieve the goals, and team structures for agents to execute the plans. An individual process only contains the portion of the team process related to the agent, which corresponds to the individual responsibilities of roles delegated to the agent. Therefore, a team process in RoB-CAST is distributed in agents' RoB-SMMs; the agents' individual processes are complementary to each other; and the agents' RoB-SMMs have a low level of overlapping (i.e., their team organizations). Our experimental results demonstrates that RoB-CAST can dramatically reduce the communication for maintaining shared mental models, support concurrent executions, and further improve team performance.

Fourth, based on RoB-SMMs, we have developed two reasoning mechanisms, including RoB-PIE and RoB-PHB, to improve team performance. Through RoB-PIE, agents in RoB-CAST can anticipate information needs of other team members and proactively provide information to them. Our experimental results demonstrate that RoB-PIE can facilitate information exchange across the boundaries of plan invocations (at top level). Through RoB-PHB, agents in RoB-CAST can identify helps needs of other team members and proactively provide helping behaviors (by initializing a role-based plan in our implementation) to cover the help needs. Our experimental results demonstrate that RoB-PHB can facilitate helping behaviors among agents and consequently improve team performance.

VII.2 Future Work

There are several interesting issues related to teamwork that are still unresolved and may be included in RoB-MALLET and RoB-CAST. The research results presented here also touch on the areas of research in other teamwork architectures.

VII.2.1 Helping Behaviors

The mechanism of role-based proactive helping behavior currently focuses on identifying help needs, searching plans as helping behaviors and forming teams to actually execute the found plans. Several related issues have not been captured in this mechanism:

- Current mechanism captures two types of help needs: 1) help needs for backing up failures, and 2) help needs for reaching the prerequisites of goals. Our formalization of RoB-PHB only treats a failure as one of an entire team. For example, if two agents execute a plan and they fail to finish the plan because just one of them loses a certain ability (this is represented by termination conditions), our mechanism just backs up the failure by a team invoking a role-based plan. An extension can be made to recognize which agent causes the failure, and to have another able agent to replace that agent. Particularly, our mechanism of role-based plans can pave a way for this extension: agents can recognize which role caused the failure and let another able agent take over the role. Such extension preserves what the agents have done before the failure occurs and resumes the rest after another able agent replaces the agent that failed. So it should be able to improve team performance better than the current mechanism. More types of helping behaviors have been discussed by Castelfranchi in [15], including literal help, overhelp, critical help, overcritical help and hyper-critical help.
- In current mechanism, agents just execute a found plan to provide helping behaviors. The mechanism does not consider the impacts of doing so on the agents. Apparently, doing so benefits the agents who need helps. However, it may hurt what other agents are doing. For example, helping providers may have

other higher priority of tasks; or the plan used for helping may lead to some effects which is reverse to the goals which some agents are achieving. Also, more than one plan can be used to provide helping behaviors and the costs to execute the plans are different. An extension can be made to weight the costs, side effects and benefits of these plans and apply theoretic decision-making to decide whether to provide helping behaviors or which plan is used to pr provide helping behaviors.

- As we discussed in Chapter VI, help needs may diminish or even disappear after agents start plans to provide helping behavior. The formalization of RoB-PHB does not capture this characteristic. Agents do not monitor help needs after they start plans to cover the help needs. So the agents cannot stop the plans even though the help needs do not exist any more. An extension can be made to capture this characteristic. Also, as suggested in Chapter VI, additional extension to RoB-MALLET may be made to specify the motivations why agents start plans. For helping behavior, the motivations are help needs. So, once the help needs do not exist, the agents lose the motivations to execute the plans for covering the help needs and thus stop executing the plans.

VII.2.2 Planning

In this dissertation, we have explained how role-based plans, together with operators, can be used in planning algorithms. However, we just presented an algorithm to search for a role-based plan to achieve a goal without considering operators or decomposing the goal to a sequence of states. In the future research, we will extend this algorithm to include operators and goal decomposition as forward/backward planning algorithms do.

Moreover, planning with existing team plans (i.e., existing role-based plans in our case) is essentially related to new research issue emerging with multi-agents systems: planning for a team of agents, instead just for a single agent as traditional planning algorithms do. The search space in traditional planning algorithms for a single agent is just one dimension: the set of operators. However, the search space in planning

algorithms that use role-based plans becomes two dimensions: 1) the set of operators and role-based plans, and 2) the set of agents that execute operators/plans. Consequently, the total search space becomes the multiplicity of these two dimensions and thus its complexity increases dramatically.

In this dissertation, we suggested that a role-based plan be used as an operator in forward/backward state-space search after it is determined that there is an admissible team assignment for agents to invoke the plan. This method actually breaks planning into two separate searches (A CSP algorithm is a searching algorithm too). Time complexity is always a major concern of both planning and CSP algorithms. So, various heuristics are applied to speed up searching in planning algorithms and CSP algorithms. Although we can apply heuristics to those two separate searches, the heuristics are only on one. Consequently, We miss the chance to develop heuristics on both dimensions. Intuitively, heuristics on both dimensions have more chance to speed up searching given the search space is the multiplicity of the two dimensions. In the future research, we will combine those two searches into one and develop a heuristics on the two dimensions.

VII.2.3 Time

In reality, time is an important aspect of teamwork. For example, suppose Alice and Bob plan to play table tennis in a recreation center. However, before they play table tennis, Alice wants to do step training and Bob wants to swim. They decide that, 1) they meet each other at table A at 8:30pm; 2) if one does not show up before 8:45 pm, the other knows that he/she got tired and left already, and does not need to wait more time. Another example is a consumer-producer system, in which a consumer needs to wait 10ms after data have been put in the buffer before trying to take it out, and a producer cannot wait more than 200ms before putting newly created data in the buffer.

The current versions of MALLET and RoB-MALLET cannot express this knowledge because time was not considered when MALLET and RoB-MALLET were developed. The only construct related to time in MALLET and RoB-MALLET is false precondition handling in WAIT modes, for example, the precondition of operator movein used in experiment 2 of Chapter VI.

```
(ioper movein (?mix ?miy)
(pre-cond (nowumpus ?mix ?miy):if-false WAIT 10000)
)
```

This statement does not express any information about when operator movein is executed and how long it lasts. In the example of Alice and BoB's exercise, action "wait" should only be taken between 8:30 pm and 8:45 pm. In the example of consumer-producer, the action of taking data lasts more than 10ms and the action of putting data lasts less than 200ms.

In the future research, RoB-MALLET can be extended to express the knowledge related to time, including when actions are taken and how long they last. For example, duration (or a range of duration) can be included in the specification of an operator to specify how long the execution of the operator takes. Another example is, DO construct may be extended to included a time when (or before which) an invocation take place, and the semantics of DO construct may be extended to be "if the invocation does not take place at the time (or before the time), then …". The hardest part of this issues is we need investigate all kinds of process knowledge related to time and then develop a set of constructs which allows us to express all the kinds of process knowledge related to time. As long as the syntax and semantics of the constructs are decided, corresponding extension of the RoB-CAST-PN can be made to represent knowledge about time in individual process. For example, RoB-CAST-PN can be extended to include the technologies of Time Petri Nets [66] and Timed Petri Nets [81].

The first two extensions that we proposed above can be naturally added into our RoB-CAST without reconstructing our RoB-MALLET and RoB-CAST. As stated above, the third extension requires modifications of RoB-MALLET and further of RoB-CAST. However, the current syntax of RoB-MALLET can be viewed a special case of knowledge irrelevant to time, and its semantics can be kept unchanged. Moreover, the methods, including role-based plans, task decomposition, task delegation, the overall schema of RoB-SMMs, and the reasoning mechanisms based on RoB-SMMs would not be changed, or only changed slightly.

REFERENCES

[1] M. Barbuceanu, The role of obligations in multiagent coordination, Applied Artificial Intelligence 13 (1999) 11-38.

[2] M. Barbuceanu and M. S. Fox, COOL: a language for describing coordination in multi agent systems, in: 1st International Conference on Multi-Agent Systems (ICMAS '95), San Francisco, CA, 1995.

[3] M. Barbuceanu, T. Gray, and S. Mankovski, Coordinating with obligation, in: 2nd International Conference on Autonomous Agents (Agents '98), Minneapolis, MN, 1998.

[4] B. J. Biddle and E. J. Thomas, Role Theory: Concepts and Research, R.E. Krieger Publishing Co., New York, NY, 1979.

[5] M. E. Bratman, Intention, Plans, and Practical Reasons, Harvard University Press Cambridge, MA, 1987.

[6] M. E. Bratman, What is intention? in: Intentions in Communications, P. R. Cohen, J. Morgan, and M. E. Pollack (Eds.), MIT Press, Cambridge, MA, 1990, pp. 15-32.

[7] M. E. Bratman, Faces of Intention, Cambridge University Press, New York, NY, 1999.

[8] M. E. Bratman, D. J. Israel, and M. E. Pollack, Plans and resources bounded practical reasoning, Computational Intelligence 4(4) (1988) 349-355.

[9] S. Brehm and S. M. Kassin, Social Psychology, Second Edition, Houghton Mifflin, Boston, MA, 1993.

[10] P. Busetta, R. Ronnquist, A. Hodgson, A. Lucas, JACK intelligent agents - components for intelligent agents in Java, Technical Report TR9901, Agent Oriented Software, Melbourne, Australia, 1999.

[11] J. A. Cannon-Bowers, E. Salas, Reflections on shared cognition, Journal of Organizational Behavior 22 (2001) 195-202.

[12] J. A. Cannon-Bowers, E. Salas, and S. A. Converse, Shared mental models in expert team decision making, in: Individual and Group Decision-Making: Current Issues, J. Castellan, (Ed.), Lawrence Erlbaum Associates, Hillsdale, NJ, 1993.

[13] S. Cao, R. A. Volz, T. R. Ioerger, Y. Zhang, and J. Yen, Role-based and agent-oriented teamwork modeling, in: International Conference on Artificial Intelligence (IC-AI'2002), Las Vegas, NV, 2002.

[14] S. Cao, R. A. Volz, J. Johnson, M. Nanjanath, J. Whetzel, and D. Xu, Development of a distributed multi-player computer game for scientific experimentation and development of computer games, The Electronic Library - The Int. J. for the Applications of Technology in Information Environments 22 (1) (2004) 43-54.

[15] C. Castelfranchi, Modeling social action for AI agents, Artificial Intelligence 103 (1995) 157-182.

[16] C. Castelfranchi and R. Falcone, From task delegation to role delegation, in: Lecture Notes in Artificial Intelligence, M. Lenzerini (Ed.), Springer-Verlag, Berlin, Germany, 1997.

[17] A. Cesta, M. Miceli, and P. Rizzo, Help under risky conditions: robustness of the social attitude and system performance, in: 2nd International Conference on Multi-Agent Systems (ICMAS'96), Menlo Park, CA, 1996.

[18] W. Chainbi, A. Ben-Hamadou, and M. Jmaiel, A belief-goal-role theory for multiagent systems, International Journal of Pattern Recognition and Artificial Intelligence 15 (3) (2001) 435-450.

[19] W. Chainbi, C. Hanachi, and C. Sibertin-Blanc, The multi-agent prey/predator problem: a petri net solution, in: CESA '96 IMACS Multiconference Computational Engienering in Systems Applications (CESA), Lille, France, 1996.

[20] P. R. Cohen and H. J. Levesque, Intention is choice with commitment, Artificial Intelligence 42 (2) (1990) 213-261.

[21] P. R. Cohen and H. J. Levesque, Performatives in a rationally based speech act theory, in: 28th Annual Meeting of the Association for Computational Linguistics, University of Pittsburgh, Pittsburgh, PA, 1990.

[22] P. R. Cohen and H. J. Levesque, Teamwork, Nous: Special Issue on Cognitive Science and Artificial Intelligence 25 (4) (1991) 487-512.

[23] P. R. Cohen, H. J. Levesque, and I. Smith, On team formation, in: Contemporary Action Theory, Synthese, J. H. a. R. Tuomela (Eds.), Synthese, Kluwer Academic Publishers, Dordrecht, Netherlands, 1997.

[24] M. D. Coovert and K. McNelis, Team decision making and performance: a review and proposed modeling approach employing petri nets, in: Teams: Their

Training and Performance, R. W. Swezey and E. Salas (Eds.), Ablex Pub Corp, Norwood, NJ, 1992, pp. 247-280.

[25] M. d'Inverno, D. Kinny, M. Luck, and M. Wooldridge, A formal specification of dMARS, Intelligent Agents Iv, 1365 (1998) 155-176.

[26] M. d'Inverno, M. Luck, M. Georgeff, D. Kinny, and M. Wooldridge, The dMARS architechure: a specification of the distributed multi-agent reasoning system, Autonomous Agents and Multi-Agent Systems (2004) 5-53.

[27] Fagin, R., Halpern, J.Y., Moses, Y., and Vardi, M.Y., Reasoning about Knowledge, The MIT press, Cambridge, MA, 1995.

[28] X. Fan, J. Yen, M. Miller, T. Ioerger, and R. Volz, MALLET - a multi-agent logic language for encoding teamwork, IEEE Transactions on Knowledge and Data Engineering archive, Submitted.

[29] J. Ferber and O. Gutknecht, A meta-model for the analysis and design of organizations in multi-agent systems, in: International Conference on MultiAgent Systems (ICMAS 98), Paris, France, 1998.

[30] J. Ferber, O. Gutknecht, C. M. Jonker, J. Treur, and J. P. Muller, Organization models and behavioral requirements specification for multi-agent systems, in: 4th International Conference on MultiAgent Systems (ICMAS-2000), Boston, MA, 2000.

[31] M. Gelfond and V. Lifschitz, Representing action and change by logic programs, Logic Programming 17 (1993) 301-323.

[32] M. Gelfond and V. Lifschitz, Action languages, Electronic Transactions on Artificial Intelligence 2 (3) (1998) 193-210.

[33] M. Georgeff, B. Pell, M. Pollack, M. Tambe, and M. Wooldridge, The belief-desire-intention model of agency, in: Intelligent Agents V 1365, J. P. Muller, M. Singh, and A. Rao (Eds.), Springer-Verlag, Berlin, Germany, 1999, pp. 1-10.

[34] J. Giampapa, K. Sycara, Team-oriented agent coordination in the RETSINA multi-agent system, Technical Report CMU-RI-TR-02-34, Robotics Institute, Carnegie Mellon University, Pittsburgh, PA, 2002.

[35] B. Grosz, The contexts of collaboration, in: Cognition, Agency and Rationality, K. Korta, E. Sosa, and X. Arrazola (Eds.), Kluwer Press, Dordrecht, Netherlands, 1999, pp. 175-188.

[36] B. Grosz and S. Kraus, Collaborative plans for complex group actions, Artificial Intelligence 86 (2) (1996) 269-357.

[37] B. Grosz and S. Kraus, The evolution of shared plans, in: Foundations and Theories of Rational Agencies, A. Rao and M. Wooldridge (Eds.), Kluwer Academic Publishers, Dordrecht, Netherlands, 1999, pp. 227-262.

[38] L. He and T. R. Ioerger, A quantitative model of capabilities in multi-agent systems, in: International Conference on Artificial Intelligence (IC-AI'03), Las Vegas, NV, 2003.

[39] L. Hunsberger and B. J. Grosz, The dynamics of intentions in collaborative intentionality, in: Conference on Collective Intentionality IV, Sienna, Italy, 2004.

[40] W. v. d. Hoek, B. v. Linder, and J.-J. C. Meyer, A logic of capabilities, 1993, http://www.agent.ai/doc/upload/200403/hoek93_1.pdf.

[41] T. R. Ioerger, JARE menu, 2001, http://www.cs.tamu.edu/faculty/ioerger/Jare /Jare.html.

[42] T. R. Ioerger, Reasoning about beliefs, observability and information exchange in teamwork, in: 17th International Conference of the Florida Artificial Intelligence Research Society (FLAIRS'04), Miami Beach, FL, 2004.

[43] T. R. Ioerger and J. C. Johnson, A formal model of responsibilities in agent-based teamwork, International Conference on Artificial Intelligence (IC-AI'2001), Las Vegas, NV, 2001.

[44] T. R. Ioerger, R. A. Volz, and J. Yen, Modeling cooperative, reactive behaviors on the battlefield with intelligent agents, in: 9th Conference on Computer Generated Forces and Behavioral Representation, Orlando, FL, 2000.

[45] JACK teams manual, 2003, http://www.agent-software.com/shared/demosNdocs/ JACK-Teams-Manual.pdf.

[46] J. W. Janneck, Compositional petri net structures - introduction and formal definition, Technical Report 60, Computer Engineering and Networks Laboratory Swith Federal Institute of Technology (ETH), Zurich, Switzerland, 1998.

[47] N. R. Jennings, On being responsible, in: Decentralized Artificial Intelligence, E. Werner and Y. Demazeau (Eds.), North Holland Publishers, Amsterdam, Holland, 1992, pp. 93-102.

[48] N. R. Jennings, A knowledge level approach to collaborative problem solving, in: AAAI Workshop on Cooperation Among Heterogeneous Intelligent Agents, San Jose, CA, 1992.

[49] N. R. Jennings, Using GRATE to build cooperating agents for industrial control, in: IFAC/IFIP/IMACS International Symposium on Artificial Intelligence in Real Time Control, Delft, Netherlands, 1992.

[50] N. R. Jennings, Commitments and conventions: the foundation of coordination in multi-agent systems, The Knowledge Engineering Review 8 (3) (1993) 223-250.

[51] N. R. Jennings and J. R. Campos, Towards a social level characterisation of socially responsible agents, IEE Proceedings on Software Engineering 144 (1) (1997) 11-25.

[52] N. R. Jennings and E. H. Mamdani, Using joint responsibility to coordinate collaborative problem solving in dynamic environments, in: 10th National Conference on Artificial Intelligence (AAAI-92), San Jose, CA, 1992.

[53] N. R. Jennings, E. H. Mamdani, I. Laresgoiti, J. Perez, and J. Corera, GRATE: a general framework for cooperative problem solving, IEEI-BCS Journal of Intelligent Systems Engineering 1 (2) (1992) 102-114.

[54] P. N. Johnson-Laird, Mental Models, Harvard University Press, Cambridge, MA, 1983.

[55] R. Klimoski and S. Mohammed, Team mental model: construct or metaphor? Journal of Management 20 (1994) 403-437.

[56] J. Laird, A. Newell, and P. Rosenbloom, SOAR: an architecture for general intelligence, Artificial Intelligence 33 (1) (1987) 1-64.

[57] J. Langan-Fox and J. Anglim, Mental models, team mental models, and performance: process, development, and future directions, Human Factors and Ergonomics in Manufacturing 14 (4) (2004) 331-352.

[58] J. Langan-Fox, A. Worth, S. Code, and K. Langfield-Smith, Analyzing shared and team mental models, International Journal of Industrial Ergonomics 28 (2001) 99-112.

[59] H. J. Levesque, P. R. Cohen, and J. Nunes, On acting together, in: National Conference on Artificial Intelligence (AAAI '90), Boston, MA, 1990.

[60] L. L. Levesque, J. M. Wilson, and D. R. Wholey, Cognitive deivergence and shared mental models in software development project teams, Shared Cognition (Special Issue), Journal of Organizational Behavior 22 (2001) 135-144.

[61] V. Lifschitz, On the semantics of STRIPS, in: Proceedings of the 1986 Workshop of Reasoning about Actions and Plans, Los Altos, CA, 1986.

[62] V. Lifschitz, Two components of an action language, in: Annals of Mathematics and Artificial Intelligence 21, Kluwer Academic Publishers, Dordrecht, Netherlands, 1997, pp. 305-320.

[63] G. Lind, How moral is helping behavior? in: American Education Research Association (AERA), Chicago, IL, 1997.

[64] M. Ljungberg, A. Rao, G. Tidhard, D. Kinny, and E. Sonenberg, Representing and executing social plans, Australian Artificial Intelligence Institute and The University of Melbourne, Melbourne, Australia, 1992.

[65] J. E. Mathieu, T. S. Heffner, G. F. Goodwin, E. Salas, and J. A. Cannon-Bowers, The influence of shared mental models on team process and performance, Journal of Applied Psychology 85 (2) (2000) 273-283.

[66] P. Merlin, A study of the recoverability of computer systems, Ph.D Thesis, University of California, Irvine, CA, 1974.

[67] M. Miceli, A. Cesta, and P. Rizzo, Autonomous help in distributed work environments, in: ECCE7 - 7th European Conference on Cognitive Ergonomics, Bonn, Germany, 1994.

[68] D. Miller, A preliminary typology of organizational learning: synthesizing the literature, Journal of Management 22 (1996) 485-505.

[69] M. S. Miller and R. A. Volz, Training for teamwork, in: 5th World Multi-Conference on Systemics, Cybernetics and Informatics, Orlando, FL, 2001.

[70] M. S. Miller, J. Yin, T. R. Ioerger, J. Yen, and R. A. Volz, Training teams with collaborative agents, in: Proceedings of the Fifth International Conference on Intelligent Tutoring Systems, Montreal, Canada, 2000.

[71] S. Mohammed and B. C. Dumville, Team mental models in a team knowledge framework: expanding theory and measurement across disciplinary boundaries, Shared Cognition (Special Issue), Journal of Organizational Behavior 22 (2001) 89-106.

[72] S. Mohammed, E. Ringseis, From cognitive diversity to cognitive consensus in group decision making: the role of inputs, processes, and outcomes, Journal of Organizational Behavior and Human Decision Processes. In press.

[73] K. L. Myers, User's guide for the procedural reasoning system, Artificial Intelligent Center, SRI International, Menlo Park, CA, 1997.

[74] J. L. Peterson, Petri Net Theory and The Modeling of Systems, Prentice-Hall, Englewood Cliffs, NJ, 1981.

[75] J. Plymale, Extensions to the jare inference engine, Technical Report TSSTI-TR-11-04, Department of Computer Science, Texas A&M University, College Station, TX, 2004.

[76] M. E. Pollack, A model of plan interferece that distinguishes between the beliefs of actors and observers, in: 24th Annual Meeting of the Association for Computational Linguisitics, New York, NY, 1986.

[77] M. E. Pollack, Plans as complex mental attitudes, in: Intentions in Communication, P. R. Cohen, J. Morgan, and M. E. Pollack (Eds.), The MIT Press, Cambridge, MA, 1990, pp. 77-105.

[78] M. E. Pollack, The uses of plans, Artificial Intelligence 57 (1) (1992) 43-69.

[79] C. O. Porter, J. R. Hollenbeck, D. R. Ilgen, A. P. J. Ellis, B. J. West, and H. Moon, Backing up behaviors in teams: the role of personality and legitimacy of need, Applied Psychology I (2002) 391-403.

[80] D. Pynadath and M. Tambe, The commnicative multiagent team decision problem: analyzing teamwork theories and models, Journal of AI Research (JAIR) 16 (2002) 389-423.

[81] C. Ramchandani, Analysis of asynchronous concurrent systems by timed Petri Nets, Technical Report TR-120, Massachusetts Institute of Technology, Cambridge, MA, 1974.

[82] A. S. Rao and M. P. Georgeff, Modeling rational agents within a bdi-architecture, in: Proceedings of the 2nd International Conference on Principles on Knowledge Representation and Reasoning (KR'91), Cambridge, MA, 1991.

[83] A. S. Rao and M. P. Georgeff, BDI agents: from theory to practice, in: 1st International Conference on Multi-Agent Systems (ICMAS-95), San Francisco, CA, 1995.

[84] A. S. Rao, M. P. Georgeff, E. A. Sonenberg, Social plans: a preliminary report, Technical Report 24, Australian Artificial Intelligence Institute, Melbourne, Australia, 1992.

[85] W. B. Rouse, J. A. Cannon-Bowers, and E. Salas, The role of mental models in team performance in complex systems, IEEE Transactions on System, Man and Cybernetics 22 (6) (1992) 1296-1308.

[86] S. Russell and P. Norvig, Artificial Intelligence - A Modern Approach, Prentice Hall, Upper Saddle River, NJ, 1995.

[87] E. Salas, D. Rozell, B. Mullen, and J. E. Driskell, The effect of team building on performance, an integration, Small Group Research 30 (5) (1999) 309-329.

[88] J. R. Searle, Collective intentions and actions, in: Intentions in Communication, P. R. Cohen, J. Morgan, and M. E. Pollack (Eds.), MIT Press, Cambridge, MA, 1990, pp. 401-415.

[89] W. L. Shebilske, R. A. Volz, K. M. Gildea, J. W. Workman, M. Nanjanath, S. Cao, and J. Whetzel, Revised space fortress: a validation study, Behavior Research Methods, Instruments, and Computers. To appear 2005.

[90] O. Shehory, K. Sycara. The RETSINA communicator, in: Proceedings of Autonomous Agents and Multi-Agent Systems, Barcelona, Catalonia, Spain, 2000.

[91] H.A. Simon, and Associates, Decision making and problem solving, in: Report of the Research Briefing Panel on Decision Making and Problem Solving, the National Academy of Sciences, National Academy Press, Washington, DC. 1986.

[92] I. A. Smith and P. R. Cohen, Toward a semantics for an agent communications language based on speech-acts, in: Annual Meeting of the American Association for Artificial Intelligence (AAAI-96), Portland, OR, 1995.

[93] K. A. Smith-Jentsch, G. E. Campbell, D. M. Milanovich, and A. M. Reynolds, Measuring teamwork mental models to support training needs assessment, development, and evaluation: two empirical studies, shared cognition (special issue), Journal of Organizational Behavior 22 (2001) 179-194.

[94] E. Sonenberg, G. Tidhar, E. Werner, D. Kinny, M. Ljungberg, and A. Rao, Planned team activity, Technical Notes 26, Australian Artificial Intelligence Institute, Melbourne, Australia, 1992.

[95] G. Stasser, W. Titus, Pooling of unshared information in group decision making: biased information sampling during discussion, Journal of Personality and Social Psychology 48 (1985) 1467-1478.

[96] P. Stone and M. Veloso, Task decomposition, dynamic role assignment, and low-bandwidth communication for real-time strategic teamwork, Artificial Intelligence 110 (2) (1999) 241-273.

[97] R. J. Stout, J. A. Cannon-Bowers, E. Salas, and D. M. Milanovich, Planning, shared mental models, and coordinated performance: an empirical link is established, Human Factors 41 (1) (1999) 61-71.

[98] D. G. Sullivan, A. Glass, B. J. Grosz, and S. Kraus, Intention reconciliation in the context of teamwork: an initial empirical investigation, in: Cooperative Information Agents III, Lecture Notes in Artificial Intelligence, vol. 1652, M. Klusch, O. M. Shehory, and G. Weiss (Eds.), Springer-Verlag, Berlin, Germany, 1999, pp. 149-162.

[99] D. G. Sullivan, B. J. Grosz, and S. Kraus, Intention reconciliation by collaborative agents, in: 4th International Conference on Multi-Agent Systems (ICMAS-2000), Boston, MA, 2000.

[100] K. Sycara, M. Paolucci, M. v. Velsen, and J. Giampapa, The RETSINA MAS infrastructure, Technical Report CMU-RI-TR-01-05, Robotics Institute, Carnegie Mellon University, Pittsburgh, PA, 2001.

[101] M. Tambe, Tracking dynamic team activity, in: National Conference on Artificial Intelligence (AAAI), Portland, OR, 1996.

[102] M. Tambe, Agent architectures for flexible, practical teamwork, in: 14th National Conference on Artificial Intelligence, Providence, RI, 1997.

[103] M. Tambe, Towards flexible teamwork, Journal of Artificial Intelligence Research 7 (1) (1997) 83-124.

[104] M. Tambe, Agent architectures for flexible, practical teamwork, in: National Conference on Artificial Intelligence (AAAI-97), Providence, RI, 1997.

[105] M. Tambe and P. S. Rosenbloom, Architectures for agents that track other agents in multi-agent worlds intelligent agents, in: M. Wooldridge, J. P. Muller, and M. Tambe (Eds.), Intelligent Agents Volume II, Lecture Notes in Artificial Intelligence, Springer-Verlag, Berlin, Germany, 1996, pp. 156-170.

[106] G. Tidhar, Team-oriented programming: preliminary report, Technical Note 41, Australian Artificial Intelligence Institute, Melbourne, Australia, 1993.

[107] G. Tidhar, A. Rao, M. Ljungberg, D. Kinny, and E. Sonenberg, Skills and capabilities in real-time team formation, Technical Note 27, Australian Artificial Intelligence Institute, Melbourne, Australia, 1992.

[108] G. Tidhar, A. S. Rao, and E. A. Sonenberg, Guided team selection, in: Proc. of the 2nd International conference of Multi-agent systems, Menlo Park, CA, 1996.

[109] R. Tuomela, On the structural aspects of collective action and change by logic programs, Theory and Decision 32 (2) (1992) 269-357.

[110] R. Tuomela, philosophy and distributed artificial intelligence: the case of joint intention, in: Foundations of Distributed Artificial Intelligence, N. Jennings and G. O'Hare (Eds.), Wiley Press, New York, NY, 1992, pp. 487-503.

[111] R. Tuomela, Collective and joint intention, Mind and Society 1 (2) (2000) 39-69.

[112] R. Tuomela, Collective intentionality and social agents, in: AI Conference IMF, Toulouse, France, 2001.

[113] R. Tuomela and K. Miller, We-intention, Philosophical Studies 53 (1988) 367-389.

[114] R. Tuomela and G. Sandu, Joint action and group action made precise, Synthese 105 (1995) 319-345.

[115] M. Villano, Probabilistic student models: bayesian belief networks and knowledge space theory, Intelligent Tutoring Systems, Montreal, Canada, 1992.

[116] R. A. Volz, J. C. Johnson, S. Cao, M. Nanjanath, J. Whetzel, T. R. Ioerger, B. Raman, W. L. Shebilske, and D. Xu, Fine-grained data acquisition and agent oriented tools for distributed training protocol research: revised space fortress, accepted for BRMIC Electronic Archive. To appear 2005.

[117] Webster's revised unabridged dictionary, MICRA, Inc., Plainfield, NJ, 1998.

[118] D. M. Wegner, Transactive memory: a contemporary analysis of the group mind, in: Theories of Group Behavior, I. B. Mullen, G. R. Goethal (Eds.), Spring-Verlag, New York, NY, 1987, pp. 185-208.

[119] M. Wooldridge, Coherent social action, in: 11th European Conference on AI (ECAI-94), Amsterdam, Holland, 1994.

[120] M. Wooldridge and N. R. Jennings, towards a theory of cooperative problem solving, in: Proc. Modelling Autonomous Agents in a Multi-Agent World (MAAMAW-94), Odense, Denmark, 1994, pp. 15-26.

[121] M. Wooldridge and N. Jennings, Intelligent agents: theory and practice, Knowledge Engineering Review 10 (2) (1995) 115-152.

[122] D. Xu, R. A. Volz and T. R. Ioerger, Generating parallel plans based on planning graph analysis of predicate/transition nets, in: Proc. of the 2002 International Conference on Artificial Intelligence (IC-AI'02), Las Vegas, NV, 2002.

[123] D. Xu, R. A. Volz, T. R. Ioerger, and J. Yen, Modeling and analyzing multi-agent behaviors using predicate/transition nets, The International Journal of Software Engineering and Knowledge Engineering 13 (1) (2003) 103-124.

[124] J. Yen and X. Fan, The semantics of proactive communication acts among team-based agents, in: IEEE International Conference on Tools with Artificial Intelligence (ICTAI), University Park, PA, 2002.

[125] J. Yen, X. Fan, and R. A. Volz, On proactive delivery of needed information to teammates, in: Autonomous Agents and Multi-Agent Systems 2002 Workshop on Teamwork and Coalition Formation, Bologna, Italy, 2003.

[126] J. Yen, X. Fan, and R. A. Volz, Proactive communications in agent teamwork, in: Advances in Agent Communication, F. Dignum (Ed.), Springer-Verlag, Berlin, Germany, 2004, pp. 271-290.

[127] J. Yen, J. Yin, T. R. Ioerger, M. S. Miller, D. Xu, and R. A. Volz, CAST: collaborative agents for simulating teamwork, in: 17th International Joint Conference on Artificial Intelligence (IJCAI'2001), Seattle, WA, 2001.

[128] J. Yin, A multi-agent framework for simulating proactive teamwork, Ph.D Thesis, Texas A&M University, College Station, TX, 2001.

[129] J. Yin, M. S. Miller, T. R. Ioerger, J. Yen, and R. A. Volz, A knowledge-based approach for designing intelligent team training systems, in: 4th International Conference on Autonomous Agents, Barcelona, Spain, 2000.

[130] M. Yokoo, E. H. Durfee, T. Ishida, and K. Kuwabara, The distributed constraint satisfaction problem: formalization and algorithms, IEEE Transactions on Knowledge and Data Engineering Archive 10 (5) (1998) 673-685.

[131] Y. Zhang, R. A. Volz, T. R. Ioerger, S. Cao, and J. Yen, Proactive information exchange during team cooperation, in: International Conference on Artificial Intelligence (IC-AI'2002), Las Vegas, NV, 2002.

APPENDIX A

PositionName ::= <IDENTIFIER>

RoleVariableName ::= <VARIABLE>

RoleTeamName ::= <IDENTIFIER>

PlanName ::= <IDENTIFIER>

OperName ::= <IDENTIFIER>

PlanOrOperName ::= <IDENTIFIER>

AgentName ::= <IDENTIFIER>

TeamName ::= <IDENTIFIER>

AgentOrTeamName ::= <IDENTIFIER>

RoleName ::= <IDENTIFIER>

VariableListOpt ::= "(" <VARIABLE> ")"*

ConstraintsList ::= "(" "CONSTRAINTS" Cond+ ")"

PositionDef ::= "(" "POSITION" PositionName "(" OperName ")" ")"*

RoleTeamDef ::= "(" "ROLETEAM" RoleTeamName? "(" RoleDef+ ")" ")"

RoleList ::="(" RoleName+ ")"

Invocation ::= "(" PlanOrOperName (<IDENTIFIER> | <VARIABLE>) ")"*

AgentDef ::= "(" "AGENT" AgentName ")"

TeamDef ::= "(" "TEAM" TeamName ("(" AgentName+ ")")? ")"

MemberOf ::= "(" "MEMBEROF" AgentName (TeamName | "(" (TeamName)+ ")") ")"

Pred ::= "(" (<IDENTIFIER> | <NOT> | <EQ> | <LT> | <GT> | <LE> | <GE>) (<IDENTIFIER> | <VARIABLE> | Pred) ")"*

AssertDef ::= "(" "ASSERT" TeamName? Pred+ ")"

RuleDecl ::= "(" "RULE" Pred+ ")"

LoadDecl ::= "(" "LOAD" <IDENTIFIER> ")"

RetractDef ::= "(" "RETRACT" TeamName? Pred+ ")"

GoalDef ::= "(" "GOAL" AgentOrTeamName Pred+ ")"

Start ::= "(" "START" AgentOrTeamName Invocation ")"

CapabilityDef ::="(" "CAPABILITY" (AgentName | "("AgentName+ ")") "("OperName+ ")" ")"

RoleDef ::= "(" "ROLE" "("RoleName+ ")" PositionName ")"

PlaysRole ::= "(" "PLAYSROLE" AgentName (RoleName | "(" RoleName+ ")") ")"

FulfilledBy ::= "(""FULFILLEDBY" RoleName (AgentName | "("AgentName+ ")") ")"

PreConditionList ::= "(" "PRECOND" Pred+ PreConditionOption? ")"

PreConditionOption ::= ":IF-FALSE" ("FAIL" | "WAIT" (<INTEGER> | <VARIABLE>)? | "ACHIEVE")

EffectsList ::= "(" "EFFECTS" Pred+ ")"

TermConditionsList ::= "(""TERMCOND" ("SUCCESS"|"FAILURE")? Pred+ ")"

IOperDef ::= "(" "IOPER" OperName VariableListOpt PreConditionList? EffectsList? ")"

TOperDef ::= "(" "TOPER" OperName VariableListOpt ("AND" | "OR" | "XOR")?PreConditionList? EffectsList? ")"

PlanDef ::= "(" "PLAN" PlanName VariableListOpt RoleTeamDef ConstraintsList? PreConditionList? EffectsList? TermConditionsList? "(" "PROCESS" MalletProcess ")" ")"

ByWhomSpec ::= RoleName | RoleVariableName | "(" (RoleName | RoleVariableName)+ ")"

MalletProcess ::= SeqMalletProcess
| ParMalletProcess
| BranchMalletProcess
| LoopMalletProcess
| ForEachMalletProcess
| ForAllMalletProcess

| ChoiceMalletProcess

| JoinDoMalletProcess

| DoMalletProcess

| GoalMalletProcess

| SelectMalletProcess

| RoleAssertMalletProcess

| RoleRetractMalletProcess

SeqMalletProcess ::= "(" "SEQ" MalletProcess+ ")"

ParMalletProcess ::= "(" "PAR" MalletProcess+ ")"

BranchMalletProcess ::= "(" "IF" "(" "COND" Pred+ ")" MalletProcess MalletProcess? ")"

LoopMalletProcess ::= "(" "WHILE" "(""COND" Pred+ ")" MalletProcess ")"

ForEachMalletProcess ::= "(" "FOREACH" "(""COND" Pred+ ")" MalletProcess ")"

ForAllMalletProcess ::= "(" "FORALL" "(" "COND" Pred+ ")" MalletProcess ")"

ChoiceMalletProcess ::= "(" "CHOICE" (MalletProcess)+ ")"

JoinDoMalletProcess ::= "(" "JOINTDO" ("AND" | "OR" | "XOR")? MalletProcess+ ")"

DoMalletProcess ::= "(" "DO" ByWhomSpec Invocation ")"

GoalMalletProcess ::= "(" "ACHIEVE" ByWhomSpec Pred+ ")"

SelectMalletProcess ::= "(" "SELECT" RoleVariableName RoleList ConstraintsList ")"

RoleAssertMalletProcess ::= "(" "ASSERT" ByWhomSpec Pred+ ")"

RoleRetractMalletProcess ::= "(" "RETRACT" ByWhomSpec Pred+)"

APPENDIX B

For the purpose of recursively handling MALLET constructs, connector nodes are created when generating team responsibility as shown in the above algorithms. The connector nodes may function as flow controls. For example, Loopback and BranchEnd connector nodes in Figure 11 help express flow controls of WHILE and IF constructs. However, redundant connector nodes may be used for easy translations the algorithms described in Sec. IV.4.3. The redundant connector nodes can be reduced.

As the way of generating team responsibility by the algorithms in previous sections, the links connecting to a connector node are either dependency or guard links; and the output links are dependency links only. If a connector node is not a special type of connector node, such as Loopback or BranchEnd connector node, it can be reduced in two general situations:

1. a connector node with only one input link, either dependency or guard link. In the left part of Figure 66, *Connector* has only one input link *L*. It can be reduced by removing *Connector* and *L* and then connecting all of its output links to the predecessor node *Pred* as shown in the right part of Figure 66. If L is a guard link, the output links of Connector1 become guard links labeled with the same logic guard as L;

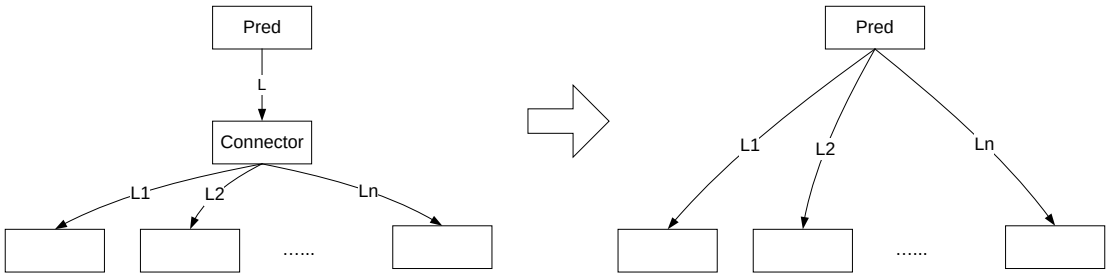

Figure 66. Reduction of a connector with only one input link.

2. a connector node with only one output link. In the left part of Figure 67, *Connector* has only one input link L. It can be reduced by removing *Connector*

and *L* and then connecting all of its input links to the successor node *Succ* as show in the right part of Figure 67.

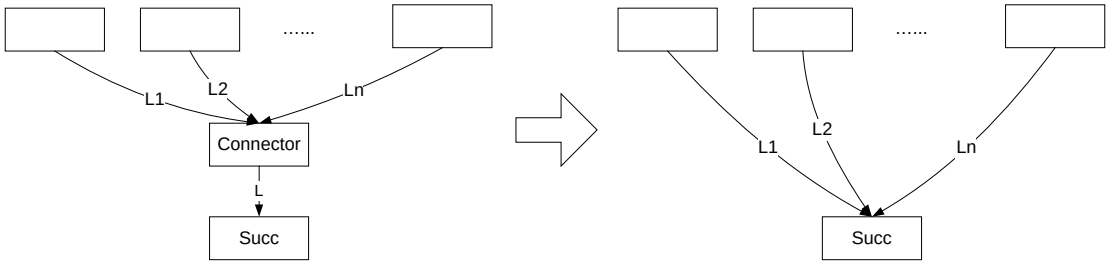

Figure 67. Reduction of a connector with only one output link.

To reduce all redundant connector nodes in a responsibility graph, all nodes in the responsibility graph can be visited by a Depth-First Search algorithm. The above strategies are applied to each node to check if the node is redundant and reduce the node corresponding if the node is redundant.

APPENDIX C

This appendix is the RoB-MALLET code representing operators used in Experiment 1.

```
;WUMPUS DOMAIN OPERATORS
;Operator move is used to move its performer to a non-wumpus square
;adjacent to its current square. The adjacent squares closer to its
;performer's target square (the target square is represented by a
;belief (targetloc r x y))have higher priorities of being selected. The
;performer r asserts a belief (loc r newx, newy) about its new location
;(newx, newy). Predicate (nowumpus ?x ?y) means that there is no wumpus
;at square (?x, ?y). See APPENDIX J for a more detailed discussion of
;the issues involved in the selection of this operator.
(ioper move ()
(pre-cond (nowumpus ?x ?y))
)

;Operator shoot is used to shoot to at square (?x, ?y). If there is a
;wumpus in square (?x, ?y), the wumpus is killed.
(ioper shoot (?x ?y))

;Operator selectSquareToDetect is used to select an unexamined square
;as the target square. If performer r selects square (x, y) to examine,
;assert a belief(detecting r x y).
(ioper selectSquareToDetect())


;Operator selectWumpusToKill is used to select a square with wumpus as
;the target square so as to kill the wumpus. If there is no square with
;wumpus, wait. Predicate (wumpus ?wwx ?wwy) means that there is a
;wumpus at square (?wwx, ?wwy). If the performer does not have any
;belief (wumpus ?wwx ?wwy), the performer just wait. If the performer r
;has one belief or more as (wumpus ?wwx ?wwy), the performer selects
```

```
;one of the wumpuses to kill (say the one at square (x, y) to kill) by
;asserting a belief(targetloc r x y).
(ioper selectWumpusToKill ()
  (pre-cond (wumpus ?wwx ?wwy))
)


;Operator selectGoldToCollect is used to select a square with gold as
;the target square from which to collect the gold. If there is no
;square with gold, wait.Predicate (nowumpus ?wgx ?wgy)means that there
;is a piece of gold at square (?wgx, ?wgy). If the performer does not
;have any belief (gold ?wgx ?wgy), the performer just waits. If the
;performer r has beliefs like (gold ?wgx ?wgy), the performer selects
;one piece of the gold to collect, say the piece at square (x, y), and
;asserts a belief(collecting r x y).
(ioper selectGoldToCollect ()
  (pre-cond (gold ?wgx ?wgy))
)

;Operator collect is used to collect gold at square (?x, ?y). If there
;is a piece of gold at square (?x, ?y) and the performer is at square
;(?x, ?y), the gold is collected. So, use a IF statement to make sure
;that the performer is at square (?x ?y).
(ioper collect (?cx ?cy))

;Operator sense is used to detect wumpus(es) and gold in the squares
;within radius 2 of the current square. After the operator is done, the
;result becomes the performer's belief. The result of operator sense
;could be (wumpus x y) or (nowumpus x y), and (gold x y) or (nogold x
;y). (wumpus x y) means that there is a wumpus at square (x, y) while
;(nowumpus x y) means that there is ;not. (gold x y) means that there
;is a piece of gold at square (x, y) while (nogold x y) means that
;there is not
(ioper sense ())
```

```
;Operator startClock is used to start clock for plan execution and
;start the performance collector. Operator startClock is only executed
;the first time when it is called by an agent.
(ioper startClock ())

;Operator getLock is used to get the lock so that an agent can handle
;one goal at a time, i.e., detect a square, kill a wumpus, or collect
;gold. Each agent has only one lock. If more than one role is delegated
;to an agent, they share the lock; otherwise, they do not.
(ioper getLock ())

;Operator freeLock is used to release the lock.
(ioper freeLock ())
```

APPENDIX D

This appendix is the RoB-MALLET code representing all knowledge, except the knowledge of operators, used in Experiment 1.

```
;Declare agents
(agent ag1)
(agent ag2)
(agent ag3)
(agent ag4)
(agent ag5)
(agent ag6)
(agent ag7)
(agent ag8)
(agent ag9)
(agent ag10)

;Declare agents capabilities
(capable ag1 (startClock move sense selectSquareToDetect shoot selectWumpusToKill collect selectGoldToCollect getLock freeLock))
(capable ag2 (startClock move sense selectSquareToDetect getLock freeLock))
(capable ag3 (startClock move shoot selectWumpusToKill collect selectGoldToCollect getLock freeLock))
(capable ag4 (startClock move shoot selectWumpusToKill getLock freeLock))
(capable ag5 (startClock move sense selectSquareToDetect collect selectGoldToCollect getLock freeLock))
(capable ag6 (startClock move collect selectGoldToCollect getLock freeLock))
(capable ag7 (startClock move sense selectSquareToDetect shoot selectWumpusToKill getLock freeLock))
(capable ag8 (startClock move sense selectSquareToDetect getLock freeLock))
```

```
(capable ag9 (startClock move shoot selectWumpusToKill getLock freeLock))
(capable ag10 (startClock move collect selectGoldToCollect getLock freeLock))

;Declare teams
(team T1 (ag1))
(team T2 (ag2 ag3))
(team T3 (ag4 ag5))
(team T4 (ag6 ag7))
(team T8 (ag8 ag9 ag10))

;Start tasks
;We list the statements for different formation of teams to execute
;plan scankillpick. Here, we just leave one start uncommented in the
;following. For a certain team, just uncomment the statement for the
;team and comment others.
(start T1 (scankillpick))
;(start T2 (scankillpick))
;(start T3 (scankillpick))
;(start T4 (scankillpick))
;(start T5 (scankillpick))

;Wumpus Domain Positions
(position sniffer (startClock move sense selectSquareToDetect getLock freeLock))
(position fighter (startClock move shoot selectWumpusToKill getLock freeLock))
(position carrier (startClock move collect selectGoldToCollect getLock freeLock))

;The definitions of team Plans
;Plan scankillpick is to scan the wumpus world, kill found wumpuses,
;and collect found gold.
(plan scankillpick ()
  (roleteam ((role (r1) sniffer)
```

```
           (role (r2) fighter)
           (role (r3) carrier)))
(process
   (par
      (seq
         (do r1 (startClock))
         (while (cond (eq 1 1))

            (seq
               (do r1 (selectSquareToDetect))
               (do r1 (getLock))
               (do r1 (detectTargetSquare))
               (do r1 (freeLock))
            )
         )
      )
      (seq
         (do r2 (startClock))
         (while (cond (eq 1 1))
            (seq
               (do r2 (selectWumpusToKill r2))
               (do r2 (getLock))
               (do r2 (killWumpus))
               (do r2 (freeLock))
            )
         )
      )
      (seq
         (do r3 (startClock))
         (while (cond (eq 1 1))
            (seq
               (do r3 (selectGoldToCollect r3))
               (do r3 (getLock))
               (do r3 (collectGold))
               (do r3 (freeLock))
            )
```

```
            )
         )
      )
   )
)

;Plan detectTargetSquare is to detect a target square.
(plan detectTargetSquare ()
   (roleteam ((role (sn) sniffer)))
   (process
      (if (cond (detecting sn ?DX ?DY))
         (seq
            (assert sn (targetloc sn ?DX ?DY))
            (while (cond (targetloc sn ?DX ?DY))
               (seq
                  (do sn (sense))
                  (do sn (move))
                  (if (cond (loc sn ?DX ?DY))
                     (retract sn (targetloc sn  ?DX ?DY))
                  )
               )
            )
            (retract sn (detecting sn ?DX ?DY))
         )
      )
   )
)

;Plan killWumpus is to kill the target wumpus.
(plan killWumpus ()
   (roleteam ((role (fi) fighter)))
   (process
      (if (cond (killing fi ?KX ?KY))
         (seq
            (assert fi (targetloc fi ?KX ?KY))
            (while (cond (targetloc fi ?KX ?KY))
```

```
            (seq
               (do fi (move))
               (if (cond (nextto fi ?KX ?KY))
                  (seq
                     (do fi (shoot ?KX ?KY))
                     (retract fi (targetloc fi ?KX ?KY))
                  )
               )
            )
         )
         (retract r2 (killing fi ?KX ?KY))
      )
   )
 )
)

;Plan collectGold is to collect the target gold.
(plan collectGold ()
  (roleteam ((role (ca) carrier)))
  (process
     (if (cond (collecting ca ?CX ?CY))
        (seq
           (assert ca (targetloc ca ?CX ?CY))
           (while (cond (targetloc ca ?CX ?CY))
              (seq
                 (do ca (move))
                 (if (cond (loc ca ?CX ?CY))
                    (seq
                       (do ca (collect ?CX ?CY))
                       (retract ca (targetloc ca  ?CX ?CY))
                    )
                 )
              )
           )
           (retract ca (collecting ca ?CX ?CY))
        )
```

)
)
)

APPENDIX E

This appendix is the RoB-MALLET code representing operators used in Experiment 2.

```
;WUMPUS DOMAIN OPERATORS
;Operator move is used to move its performer to a non-wumpus square
;adjacent to its current square. The adjacent squares closer to its
;performer's target square (the target square is represented by a
;belief (targetloc r x y))have higher priorities of being selected. The
;performer r asserts a belief (loc r newx, newy) about its new location
;(newx, newy). Predicate (nowumpus ?x ?y) means that there is no wumpus
;at square (?x, ?y). See APPENDIX J for a more detailed discussion of
;the issues involved in the selection of this operator.
(ioper move ()
(pre-cond (nowumpus ?x ?y))
)

;Operator randmove is used to move its performer to a non-wumpus square
;adjacent to its current square. All adjacent squares without wumpuses
;have equal priorities of being selected. The performer r asserts a
;belief (loc r newx, newy) about its new location (newx, newy).
;Predicate (nowumpus ?x ?y) means that there is no wumpus at square
;(?x, ?y). See APPENDIX J for a more detailed discussion of the issues
;involved in the selection of this operator.
(ioper randmove()
(pre-cond (nowumpus ?x ?y) :if-false FAIL)
)

;Operator movein is used` to move its performer to square (?mix ?miy).
;If there is a wumpus at square (?mix ?miy), wait until there is not
;(i.e., the wumpus is killed); or after 10000 ms, this operator does
;not occur. If the operator occurs, the performer r asserts a belief
;(loc r newx, newy) about its new location (newx, newy). Precondition
```

```
;(nowumpus (?mix ?miy) means that the condition under which operator
;movein occurs is that there is no wumpus at square ((?mix, ?miy).
(ioper movein (?mix ?miy)
(pre-cond (nowumpus ?mix ?miy):if-false WAIT 10000)
)

;Operator shoot is used to shoot to square (?x, ?y). If there is a
;wumpus at square (?x, ?y), the wumpus is killed.
(ioper shoot (?x ?y)
)

;Operator checkSurroundingWumpus is used to check if square (?x, ?y) is
;surrounded by wumpuses based on the beliefs of the performer r, not
;actual sensing.  Its beliefs may be updated by proactive information
;exchange by other agents. If r believes that there is a wumpus in
;every square adjacent to square (?x, ?y), r randomly selects one
;square (FX, FY) adjacent to square (?x, ?y) as the target square to
;move in by asserting a belief (hopemovein r FX FY) ; otherwise r does
;nothing.
(ioper checkSurroundingWumpus (?x ?y))

;Operator selectSquareToDetect is used to select an unexamined square
;as the target square. If performer r selects square (x, y) to examine,
;assert a belief(targetloc r x y).
(ioper selectSquareToDetect())

;Operator selectWumpusToKill is used to select a square with wumpus as
;the target square so as to kill the wumpus. If there is no square with
;wumpus, wait. Predicate (wumpus ?wwx ?wwy) means that there is a
;wumpus at square (?wwx, ?wwy). If the performer does not have any
;belief (wumpus ?wwx ?wwy), the performer just waits. If the performer
;r has one belief or more as (wumpus ?wwx ?wwy), the performer selects
;one of the wumpuses to kill, say the one at square (x, y) to kill, and
;asserts a belief(targetloc r x y).
(ioper selectWumpusToKill ()
  (pre-cond (wumpus ?wwx ?wwy))
```

```
)

;Operator selectGoldToCollect is used to select a square with gold as
;the target square from which to collect the gold. If there is no
;square with gold, wait.Predicate (gold ?wgx ?wgy) means that there is
;a piece of gold at square (?wgx, ?wgy). If the performer does not have
;any belief (gold ?wgx ?wgy), the performer just waits. If the
;performer does not have any belief (gold ?wgx ?wgy), the operator is
;not executed. If the performer r has one or more beliefs  (gold ?wgx
;?wgy), the performer selects one piece of the gold to collect, say the
;piece at square (x, y), and asserts a belief(targetloc r x y).
(ioper selectGoldToCollect ()
  (pre-cond (gold ?wgx ?wgy))
)

;Operator collect is used to collect gold at square (?x, ?y). If there
;is a piece of gold at square (?x, ?y) and the performer is at square
;(?x, ?y), the gold is collected. So, use a IF statement to make sure
;that the performer is at square (?x ?y).
(ioper collect (?cx ?cy))

;Operator sense is used to detect wumpus(es) and gold in the squares
;within radius 2 of the current square. After the operator is done, the
;result becomes the performer's belief. The result of operator sense
;could be (wumpus x y) or (nowumpus x y), and (gold x y) or (nogold x
;y). (wumpus x y) means that there is a wumpus at square (x,y) while
;(nowumpus x y) means that there is not. (gold x y) means that there is
;a piece of gold at square (x,y) while (nogold x y) means that there is
;not.
(ioper sense ())

;Operator startClock is used to start clock for plan execution and
;start the performance collector. Operator startClock is only executed
;the first time when it is called by an agent.
(ioper startClock ())
```

APPENDIX F

This appendix gives the RoB-MALLET code representing all knowledge, except the knowledge of operators, used in Experiment 2.

```
;Declare agents
(agent ag1)
(agent ag2)
(agent ag3)

;Declare agents capabilities
(capable ag1 (startClock move movein randmove sense
selectSquareToDetect))
(capable ag2 (startClock move movein randmove shoot
selectWumpusToKill))
(capable ag3 (startClock move movein randmove collect
selectGoldToCollect checkSurroundingWumpus))

;Declare teams
(team T (ag1 ag2 ag3))
(team T1 (ag1 ag3))
(team T2 (ag2))

;Start tasks
(start T2 (randommove))
(start T1 (scancollect))

;Wumpus Domain Positions
(position basic (startClock move movein randmove))
(position sniffer (startClock move sense selectSquareToDetect))
(position fighter (startClock move randmove shoot selectWumpusToKill))
(position carrier (startClock move movein collect selectGoldToCollect
checkSurroundingWumpus))

;The definitions of team Plans
```

```
;Plan randommove is to keep moving randomly when the performer has no
;target square to move.
(plan randommove ()
  (roleteam ((role (r1) basic)))
  (process
     (seq
        (do r1 (startClock))
        (while (cond (eq 1 1))
           (if (cond (not (targetloc r1 ?x ?y)))
              (do r1 (randmove))
           )
        )
     )
  )
)

;Plan scancollect is to scan the wumpus world and collect as much found
;gold as possible.
(plan scancollect ()
  (roleteam ((role (r1) sniffer) (role (r2) carrier)))
  (process
     (par
        (seq
           (do r1 (startClock))
           (while (cond (eq 1 1))
              (seq
                 (do r1 (selectSquareToDetect))
                 (if (cond (targetloc r1 ?X1 ?Y1))
                    (do r1 (movetoSense ?X1 ?Y1))
                 )
              )
           )
        )
        (seq
           (do r2 (startClock))
           (while (cond (eq 1 1))
```

```
              (seq
                 (do r2 (selectGoldToCollect))
                 (if (cond (targetloc r2 ?X2 ?Y2))
                    (do r2 (trytoCollect ?X2 ?Y2))
                 )
              )
           )
        )
     )
  )
)

;Plan movetoSense is to detect the target square (?msx, ?msy).
(plan movetoSense (?msx ?msy)
  (roleteam ((role (sn) sniffer)))
  (process
      (while (cond (targetloc sn ?msx ?msy))
         (seq
             (do sn (sense))
             (do sn (move))
             (if (cond (detected ?msx ?msy))
                 (retract sn (targetloc sn ?msx ?msy))
             )
         )
      )
  )
)

;Plan trytoCollect is try to collect the gold at square (?x,, ?y). To
;collect the gold, the role ca must to move inside the square (?x, ?y).
;If the square (?x, ?y) is surrounded by wumpuses, the role ca selects
;one of the squares adjacent to the square (?x, ?y) and hopes to move
;to the selected square. Or if there is a wumpus at the square (?x,
;?y), the role ca also hopes to move to the square (?x, ?y). However,
;if there is a wumpus in that square, the role ca would not move to the
;square. It waits for a period of time and then gives up. With
```

```
;proactive helping behaviors, other agents can recognize that the
;agents playing the role ca hopes to move to that square and there is a
;wumpus in that square, and then they try to provide help by invoking a
;role-based plan which leads to a condition that there is no wumpus at
;square (?x, ?y).
(plan trytoCollect (?x ?y)
  (roleteam ((role (ca) carrier)))
  (process
     (while (cond (targetloc ca ?x ?y))
        (seq
           (if (cond (nextto ca ?x ?y))
              (seq
                 (do ca (movein ?x ?y ))
                 (if (cond (loc ca ?x ?y))
                    (do ca (collect ?x ?y ))
                    (seq
                       (retract ca (gold ?x ?y))
                       (assert ca (unreachablegold ca ?x ?y))
                    )
                 )
                 (retract ca (targetloc ca ?x ?y))
              )
              (seq
                 (do ca (checkSurroundingWumpus ?x ?y))
                 (if (cond (hopemovein  ca ?FX ?FY))
                    (seq
                       (do ca (movein ?FX ?FY))
                       (if (cond (not (loc ca ?FX ?FY)))
                       ;the condition is true only movein fails
                          (seq
                             (retract ca (gold  ?x ?y))
                             (assert ca (unreachablegold ca ?x ?y))
                          )
                       )
                       (retract ca (hopemovein ca ?FX ?FY))
                       (retract ca (targetloc ca ?x ?y))
```

```
                    )
                    (do ca (move))
                 )
              )
           )
        )
     )
  )
)

;Plan kill is to kill the wumpus in square (?kx, ?ky).
(plan kill (?kx ?ky)
  (roleteam ((role (fi) fighter)))
  (effects (nowumpus ?kx ?ky))
  (process
     (seq
        (assert fi (targetloc fi ?kx ?ky))
        (while (cond (targetloc fi  ?kx ?ky))
           (if (cond (nextto fi ?kx ?ky))
              (seq
                 (do fi (shoot ?kx ?ky ))
                 (if (cond (nowumpus ?kx ?ky))
                  (retract fi (targetloc fi ?kx ?ky))
                 )
              )
              (do fi (move))
           )
        )
     )
  )
)
```

APPENDIX G

This appendix gives the code representing operators used in Experiment 3 for both RoB-CAST and CAST 3.0.

```
;WUMPUS DOMAIN OPERATORS
;Operator move is used to move its performer to a non-wumpus square
;adjacent to its current square. The adjacent squares closer to its
;performer's target square (the target square is represented by a
;belief (targetloc r x y))have higher priorities of being selected. The
;performer r asserts a belief (loc r newx, newy) about its new location
;(newx, newy). Predicate (nowumpus ?x ?y) means that there is no wumpus
;at square (?x, ?y). See APPENDIX J for a more detailed discussion of
;the issues involved in the selection of this operator.
(ioper move ()
(pre-cond (nowumpus ?x ?y))
)

;Operator shoot is used to shoot to square (?x, ?y). If there is a
;wumpus at square (?x, ?y), the wumpus is killed.
(ioper shoot (?x ?y))

;Operator selectSquareToDetect is used to select an examimed square as
;the target square. If performer r selects square (x, y) to examine,
;assert a belief(targetloc r x y).
(ioper selectSquareToDetect())

;Operator selectWumpusToKill is used to select a square with wumpus as
;the target square so as to kill the wumpus. If there is no square with
;wumpus, wait. Predicate (wumpus ?wwx ?wwy) means that there is a
;wumpus at square (?wwx, ?wwy). If the performer does not have any
;belief (wumpus ?wwx ?wwy), the performer just wait. If the performer r
;has one belief or more of the type (wumpus ?wwx ?wwy), the performer
;selects one of the wumpuses to kill (say the one at square (x, y) to
;kill by asserting a belief(targetloc r x y).
(ioper selectWumpusToKill ()
```

```
  (pre-cond (wumpus ?wwx ?wwy))
)

;Operator selectGoldToCollect is used to select a square with gold as
;the target square so as to collect the gold. If there is no square
;with gold, wait. Predicate (gold ?wgx ?wgy) means that there is a
;piece of gold at square (?wgx, ?wgy). If the performer does not have
;any belief (gold ?wgx ?wgy), the performer just wait. If the performer
;r has one belief or more as (gold ?wgx ?wgy), the performer selects
;one piece of the gold to collect, say the piece at square (x, y), and
;asserts a belief(targetloc r x y).
(ioper selectGoldToCollect ()
  (pre-cond (gold ?wgx ?wgy))
)


;Operator collect is used to collect gold at square (?x, ?y). If there
;is a piece of gold at square (?x, ?y) and the performer is at square
;(?x, ?y), the gold is collected. So, use a IF statement to make sure
;that the performer is at square (?x ?y).
(ioper collect (?cx ?cy))

;Operator sense is to detect wumpus(es) and gold in the squares within
;a radius 2 of the current square. After the operator is done, the
;result becomes the performer's belief. The result of operator sense
;could be (wumpus x y) or (nowumpus x y), and (gold x y) or (nogold x
;y). (wumpus x y) means ;that there is a wumpus at square (x, y) while
;(nowumpus x y) means that there is ;not. (gold x y) means that there
;is a piece of gold at square (x, y) while (nogold x y) means that
;there is not.
(ioper sense ())

;Operator startClock is used to start a clock for plan execution and
;start the performance collector. Operator startClock is only executed
;the first ;time when it is called by an agent.
(ioper startClock ())
```

```
;Operator updateMap is used to fetch the beliefs about the current
;square and adjacent squares (i.e., whether there is a piece of gold or
;a wumpus).
(ioper updateMap ())
```

APPENDIX H

This appendix is the RoB-MALLET code representing all knowledge, except the knowledge of operators, used in Experiment 3.

```
;Declare agents
(agent ag1)
(agent ag2)
(agent ag3)

;Declare agents capabilities
(capable ag1 (startClock updateMap move sense selectSquareToDetect))
(capable ag2 (startClock move updateMap shoot selectWumpusToKill))
(capable ag3 (startClock move updateMap collect selectGoldToCollect))

;Declare teams
(team T (ag1 ag2 ag3))

;Start tasks
(start T (wumpusworld))

;Wumpus Domain Positions
(position sniffer (startClock move updateMap sense
selectSquareToDetect))
(position fighter (startClock move updateMap shoot selectWumpusToKill))
(position carrier (startClock move updateMap collect
selectGoldToCollect))

;The definitions of team Plans
;Plan wumpusworld is to scan the wumpus world, kill found wumpuses, and
;collect found gold.
(plan wumpusworld ()
  (roleteam ((role (r1) sniffer)
             (role (r2) fighter)
             (role (r3) carrier)))
```

```
(process
   (par
      (seq
         (do r1 (startClock))
         (while (cond (eq 1 1))
            (seq
               (do r1 (selectSquareToDetect))
               (if (cond (targetloc r1 ?DX ?DY))
                  (do r1 (detectTargetSqaure ?DX ?DY))
               )
            )
         )
      )
      (seq
         (do r2 (startClock))
         (while (cond (eq 1 1))
            (seq
               (do r2 (selectWumpusToKill r2))
               (if (cond (targetloc r2 ?KX ?KY))
                  (do r2 (killWumpus? KX ?KY))
               )
            )
         )
      )
      (seq
         (do r3 (startClock))
         (while (cond (eq 1 1))
            (seq
               (do r3 (selectGoldToCollect r3))
               (if (cond (targetloc r3 ?CX ?CY))
                  (do r3 (collectGold ?CX ?CY))
               )
            )
         )
      )
   )
```

```
  )
)

;Plan detectTargetSqaure is to detect square (?x, ?y).
(plan detectTargetSqaure (?x ?y)
  (roleteam ((role (sn) sniffer)))
  (process
     (while (cond (targetloc sn ?x ?y))
        (seq
           (do sn (sense))
           (par
              (do sn (move))
              (do sn (updateMap))
           )
           (if (cond (loc sn ?x ?y))
              (retract sn (targetloc sn ?x ?y))
           )
        )
     )
  )
)

;Plan killWumpus is to kill the wumpus at square (?x, ?y).
(plan killWumpus (?x ?y)
  (roleteam ((role (fi) fighter)))
  (process
     (while (cond (targetloc fi ?x ?y))
        (seq
           (par
              (do fi (move))
              (do fi (updateMap))
           )
           (if (cond (nextto fi ?x ?y))
              (seq
                 (do fi (shoot ?x ?y))
                 (retract fi (targetloc fi ?x ?y))
```

```
            )
         )
      )
   )
  )

;Plan collectGold is to collect the gold at square (?x, ?y).
(plan collectGold (?x ?y)
  (roleteam ((role (ca) carrier)))
  (process
     (while (cond (targetloc ca ?x ?y))
        (seq
           (par
              (do ca (move))
              (do ca (updateMap))
           )
           (if (cond (loc ca ?x ?y))
              (seq
                 (do ca (collect ?x ?y))
                 (retract ca (targetloc ca ?x ?y))
              )
           )
        )
     )
  )
)
```

APPENDIX I

This appendix is the MALLET code representing all knowledge, except the knowledge of operators, used in Experiment 3.

```
;Declare agents
(agent ag1)
(agent ag2)
(agent ag3)

;Declare roles;
(role sniffer (startClock move sense selectSquareToDetect updateMap))
(role fighter (startClock move shoot selectWumpusToKill updateMap))
(role carrier (startClock move collect selectGoldToCollect updateMap))

;Declare teams
(team T (ag1 ag2 ag3))

;Declare capabilities
(fulfilled-by sniffer (ag1))
(fulfilled-by fighter (ag2))
(fulfilled-by carrier (ag3))

;Declare tasks;
(start T (wumpusworld ag1 ag2 ag3))

;MALLET DOMAIN PLANS
;Plan wumpusworld is to scan the wumpus world, kill found wumpuses, and
;collect found gold.
(plan wumpusworld (?sn ?fi ?ca)
  (process
     (par
        (seq (do ?sn (startClock))
        (while (cond (eq 1 1))
           (seq
              (do ?sn (selectSquareToDetect))
```

```
            (if (cond (targetloc ?sn ?DX ?DY))
               ( do ?sn (detectTargetSqaure ?sn ?DX ?DY))
            )
         )
      ))
      (seq (do ?fi (startClock))
      (while (cond (eq 1 1))
         (seq
            (do ?fi (selectWumpusToKill ?fi))
            (if (cond (targetloc ?fi ?KX ?KY))
               ( do ?fi (killWumpus ?fi ?KX ?KY))
            )
         )
      ))
      (seq (do ?ca (startClock))
      (while (cond (eq 1 1))
         (seq
            (do ?ca (selectGoldToCollect ?ca))
            (if (cond (targetloc ?ca ?CX ?CY))
               ( do ?ca (collectGold ?ca ?CX ?CY))









            )
         )
      ))
   )
  )
)

;Plan detectTargetSqaure is to detect square (?x, ?y).
(plan detectTargetSqaure (?sn ?x ?y)
  (process
     (while (cond (targetloc ?sn ?x ?y))
```

```
        (seq
           (do ?sn (sense))
           (par
              (do ?sn (move))
              (do ?sn (updateMap))
           )
           (if (cond (loc ?sn ?x ?y))
              (do ?sn (retract (targetloc ?sn  ?x ?y)))
           )
        )
     )
  )
)

;Plan killWumpus is to kill the wumpus at square (?x, ?y).
(plan killWumpus (?fi ?x ?y)
  (process
     (while (cond (targetloc ?fi ?x ?y))
        (seq
           (par
              (do ?fi (move))
              (do ?fi (updateMap))
           )
           (if (cond (nextto ?fi ?x ?y))
              (seq
                 (do ?fi (shoot ?x ?y))
                 (do ?fi (retract (targetloc ?fi  ?x ?y)))
              )
           )
        )
     )
  )
)

;Plan collectGold is to collect the gold at square (?x, ?y).
(plan collectGold (?ca ?x ?y)
```

```
  (process
     (while (cond (targetloc ?ca ?x ?y))
        (seq
           (par
              (do ?ca (move))
              (do ?ca (updateMap))
           )
           (if (cond (loc ?ca ?x ?y))
              (seq
                 (do ?ca (collect ?x ?y))
                 (do ?ca (retract (targetloc ?ca  ?x ?y)))
              )
           )
        )
     )
  )
)
```

APPENDIX J

There are a number of considerations that gave rise to the particular form of `move` and `randmove` operators used in experiment 1, 2 and 3. As these may be of some interest, they are explored briefly here.

Basically, the issues revolve around the capabilities of the underlying CAST/RoB-CAST architectures and the desire to be able to meaningfully compare execution between the architectures. There are two principal issues involved: 1) expression of conditions in a manner that will trigger PIE when useful, and 2) convenience of expression of the conditions.

CAST 3.0 triggers PIE through analysis of preconditions. It is possible to extend this to trigger PIE through conditions appearing in conditional or loop statements as well. For example PSU-CAST does this. It is also an easy extension to RoB-CAST to do so. However, in order to provide a more direct comparison with CAST 3.0, these extended features were not used in RoB-CAST. Thus, proactive information exchange is triggered off preconditions only.

Next one must consider the precondition that is needed for `move` and `randmove` operators. Ideally, one would like to move to an adjacent square that is does not contain a wumpus. The carrier or fighter would thus have to wait until the sniffer identified such a location and told them about it (through PIE). Conditions of this sort, while feasible through procedural attachments in JARE, and not convenient to express. Thus, we opted to use a simple form (i.e., `(nowumpus ?x ?y)`) for the precondition and do much of the work within the operators to decide which adjacent square the performer moves to.

Finally, we note that a recent addition to the procedural attachment capability by Jesse Plymale [75] simplifies the process of including complex preconditions. However, that work was not completed until after the experiments described herein were developed.

VITA

Name: Sen Cao

Address: Department of Computer Science

Texas A&M University

College Station, TX 77843-3112

Email Address: sen.cao@gmail.com

Education: B.S., Computer Engineering, Shanghai Maritime University, 1992

M.S., Computer Engineering, Institute of Software, Chinese Academy of Sciences, 1997